S H A L

【取扱説明書】

１９９２年　7月３１日

《SHAL VERSION 1.4x 》

ＳＨＡＬとは？

- ＳＨＡＬ（Structured High Level Assembly Language ）とは、プログラミング・スタイルは高級言語であり、実行効率はアセンブラ言語と同等であることを目標にして作られた言語です。
- ＳＨＡＬは、その名前のとうり、構造化プログラミングが可能です。
- ＳＨＡＬは、又、その名前のとうり、アセンブラの特徴をそのまま利用することができます。
- ＳＨＡＬは、記述が容易で、理解し易く、保守が容易で、かつ、メモリ効率、実行効率のよい言語です。
- ＳＨＡＬは、他の言語とのインター－フェイスをとることが容易です。

ＳＨＡＬ　ｖｅｒｓｉｏｎ　ｕｐ　履歴

ｖ１．４２	'８８	４．１５	自分自身をＴｕｒｂｏＣで、コンバイルし直すとともに言語インタフェ－スをＴｕｒｂｏＣに合わせ、ＤＸＢＩＯの利用を可とする。
ｖ１．４３	'８８	６．１２	－Ｌオプションを付けないとき、正しくアセブラソ－スに翻訳されなかったバグを修正。
ｖ１．４４	'８８	８．１０	－ｉオプションの仕様変更ｉｎｃｌｕｄｅファイルをカレントディレクトリ以外の指定したディレクトリから捜せるようにする。
ｖ１．４５	'８８	８．２６	＃ｄｅｔｉｎｅで識別する名前を１５文字から約３０文字に変更。
ｖ１．５	'９２	７．３１	

目　　次

１．ＳＨＡＬの機能と特徴 ……1

２．ＳＨＡＬの起動 ……1

３．ＳＨＡＬの構成 ……2
３．１　宣言部 ……3
３．２　モジュール部 ……5
３．４　制御指令 ……11
３．５　データの参照について ……12

４．エラー　メッセージ ……15

付録Ａ：ＳＨＡＬの制限 ……18

付録Ｂ：ＳＨＡＬの変換規則 ……18

付録Ｃ：Ｃの関数の呼び方 ……22
Ｃ．１　Ｃの関数の参照法 ……22
Ｃ．２　Ｃのインターフェースに合ったＳＨＡＬの関数の作り方 ……24
Ｃ．３　Ｃで記述したデータの参照の方法 ……24
Ｃ．４　ＡＵＴＯＭＡＴＩＣ　変数の作成と使い方 ……26

付録Ｄ：８０８６命令セット ……27
Ｄ．１　概要 ……27
Ｄ．２　データ転送命令 ……28
Ｄ．３　演算、論理、シフト命令 ……29
Ｄ．４　ストリング命令 ……31
Ｄ．５　制御移動命令 ……32
Ｄ．６　制御移動命令 ……34

付録Ｅ：各レジスタの特長 ……35

付録Ｆ：リンク　情報 ……36
Ｆ．１　アラインメント ……36
Ｆ．２　組み合わせ ……36

付録Ｇ：レコード宣言・構造体宣言 ……37
Ｇ．１　レコード宣言 ……37
Ｇ．２　構造体宣言 ……38

付録Ｈ：ＰＵＳＨとＰＯＰについて ……38

１．ＳＨＡＬの機能と特徴

・　ＳＨＡＬは、ＩＦ文、ＷＨＩＬＥ文、ＦＯＲ文、ＣＡＳＥ文等を使用することにより、構造化プログラミングを行うことができます。
・　ＳＨＡＬは、複合条件式を記述することができます。
・　ＳＨＡＬは、アセンブラと同様にレジスタ、変数等を使用することができます。
・　ＳＨＡＬは、アセンブラ言語のソースプログラムを出力します。
・　ＳＨＡＬは、簡単な変換規則でアセンブラ・ソースに変換するので、デバッガ使用の際も容易にソースとオブジェクトの対応がつき、アセンブラで書いたプログラムと同じようにデバッグできます。
・　ＳＨＡＬは、Ｃ言語とのインターフェイスを自動生成することができます。
・　ＳＨＡＬは、Ｃ言語で書かれたプログラム（関数）をＣ言語と同様な語法で利用することができるので、開発効率を上げることができます。
・　ＳＨＡＬは、各種言語に対応したプログラムを容易に書くことができます。
・　ＳＨＡＬは、必要によりアセンブラ・ソースステートメントを書くことができます。

２．ＳＨＡＬの起動

コマンド・モードで次のように入力します。

```
＞ＳＨＡＬ　ファイル名　［オプション，オプション・・・］↵
```

・　ファイル名は拡張子を省略することができます。
　拡張子が省略された場合、拡張子 ．Ｓを補って、ファイル名とします。
　例）　・以下の２つの例は同じ扱いとなります。

```
＞ＳＨＡＬ　　ＳＡＭＰＬＥ↵
＞ＳＨＡＬ　　ＳＡＭＰＬＥ．Ｓ↵
```

　　・ファイル名 'ＳＡＭＰＬＥ．Ｓ' をコンパイルしたいときは、次のように入力します。

```
＞ＳＨＡＬ　　ＳＡＭＰＬＥ↵
```

・ＳＨＡＬの出力は拡張子を　．ＡＳＭとしたファイルに対して行われます。
　（例外は　－Ａオプション指定時）
・　オプションには以下のものがあり、これらはどのような順序で入力されてもよく、又省略することも可能です。大文字、小文字の区別はありません。

－Ａ文字列　　：ＳＨＡＬの出力先を変更するオプションです。
　　・文字列の最後が￥ならば、それがＳＨＡＬの出力先のパス名として処理されます。
　　・それ以外は、ＳＨＡＬの出力先を文字列で指定されたファイルに変更します。

－Ｃ　　：ＴｕｒｂｏＣとのインターフェイスを自動生成します。
　　Ｃから呼ばれる関数をプログラムする場合、又は、Ｃの関数を呼ぶ場合のいずれにおいても、指定する必要があります。

－Ｄ文字列１［，文字列２，文字列３・・・］
　　：＃ｄｅｆｉｎｅ　文字列１　０　と同義になります。
　　＃ｄｅｆｉｎｅ　文字列２　０
　　・
　　・
　　・

－Ｆ　　：モジュールの属性をＦＡＲにします。（省略時はＮＥＡＲ）－Ｉディレクトリパス名
　　：ｉｎｃｌｕｄｅ　ｆｉｌｅがカレント・ディレクトリにないとき、この与えられたパスをファイル名の先頭につけて指定されたディレクトリをさがします。このディレクトリパスは８個まで指定可能です。最後に￥は付けないで下さい。

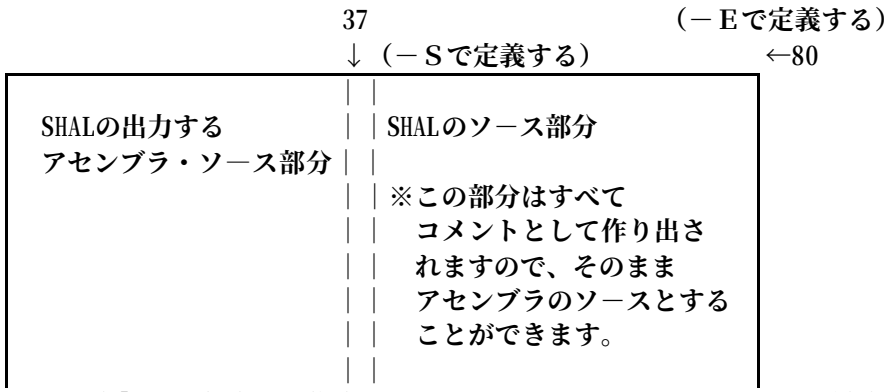
－Ｌ　　：ＳＨＡＬのソースを、アセンブラ・ソースとともに出力することを指定します。

－Ｓ［Ｎ］　　：－Ｌ　オプションを使用する場合の、アセンブラ・ソ－スの出力桁数を定義します。
　　１カラムから、Ｎで指定されるカラム位置までが、アセンブラ・ソ－ス部と見なされます。
・省略時はＮ＝３７で、３０　～　７０まで定義できます。

－Ｅ［Ｎ］　　：－Ｌ　オプションを使用する場合の、ＳＨＡＬのソ－ス・リストの出力終了位置を定義します。
・省略時はＮ＝８０で、７０　～　１３６まで定義できます。

※　プリンタ用紙のサイズに合わせて、上記－Ｓ、－Ｅオプションを指定することにより、８０桁用も１３６桁用も、きれいに出力することができます。

例）　－Ｌ　オプションを使用した場合の出力形式



－Ｍ［ファイル名］　：起動時に指定されたファイルからマクロ用ファイル（内部形式に変換される）を作ることを指定するオプションです。
・ファイル名が指定されればそのファイルから、省略されれば、ＭＡＣＲＯ．ＳＴＤという標準マクロファイルから作られます。
・内部形式に変換されたものは、ＭＡＣＲＯ．ＢＩＮというファイルで登録されます。
・－Ｍオプションが無いとき、ＭＡＣＲＯ．ＢＩＮがあれば標準マクロとして、実行に先だってマクロ展開されます。

－Ｔファイル名　　：ラベル名・手続き名のテ－ブルを指定されたファイルに出力します。

３．ＳＨＡＬの構成

ＳＨＡＬのプログラムは標準的には下記のように記述できます。

```
#INCLUDE            ファイル名
```

```
DECLARE
            共    通    部（ファイル内にあるモジュールすべてに共通な宣言部）
END

SEGMENT
            共    通    部（ファイル内にあるモジュールすべてに共通なデータ宣言部）
END

MODULE  モジュール名1 [MAIN]   [ リンク情報 ]


    DECLARE
            このモジュールで使用する宣言部
    END
    SEGMENT
            このモジュールで使用するデータ宣言部
    END
            手続き部
     PROC

     END
          ・
          ・
          ・
END （モジュール名1 ）
```

```
MODULE  モジュール名2        [リンク情報]
    DECLARE
        :
    END
        :
    PROC

    END
      ・
      ・
      ・
END （モジュール名2 ）
```

注意：　行の継続法

行１￥□

行２￥□

・

・

行ｎ

とすれば、行１～行ｎが連結されます。ただし合計長＜２５６でなければなりません。

３．１　宣言部

3.1.1　ＤＥＣＬＡＲＥ部

```
ＤＥＣＬＡＲＥ
        宣言文
ＥＮＤ
```

ＤＥＣＬＡＲＥ部では、定数、ラベル、手続き名あるいはレコ－ドや構造体などを宣言します。

・ ラベル

```
{EXTERNAL　｜　ENTRY} LABEL　{(NEAR)｜(FAR) } ラベル名
```

ＥＸＴＥＲＮＡＬ指定の場合は、外部ラベル名として参照可能になります。
ＥＮＴＲＹ指定の場合は、他のモジュ－ルから参照可能になります。
ＥＮＴＲＹ指定のでは（ＮＥＡＲ）｜（ＦＡＲ）の指定は不必要です。

・ 手続き名

```
{EXTERNAL　｜　ENTRY } PROCEDURE ［(NEAR)｜(FAR) ］手続き名
```

ＥＸＴＥＲＮＡＬ指定の場合は、外部手続き名として参照可能になります。
ＥＮＴＲＹ指定の場合は、他のモジュ－ルから参照可能になります。

・ 定数宣言

```
ＣＯＮＳＴ　定数名　＝　定数
```

プログラム内で定数名を使用した場合、定数に置き換えます。
※　レコ－ド、構造体の宣言は付録Ｇを参照して下さい。

3.1.2　ＳＥＧＭＥＮＴ部

```
SEGMENT　［(セグメント名)］[USING レジスタ名］ [リンク情報]
         :
         :　データの宣言
END
```

ＳＥＧＭＥＮＴ部は、デ－タ領域を確保する宣言で、セグメント・レジスタの指定、あるいはリンク情報を指定することができます。
メモリ領域を持つすべてのデ－タは、デ－タの宣言をする必要があります。

(1) （セグメント名）の省略時は、用いられるセグメントは次のとうりとなります。
　１．モジュ－ル内のとき・・・モジュ－ルのセグメント
　２．モジュ－ル外のとき・・・@ＤＡＴＡ
モジュ－ル内でセグメント名を省略した場合、デ－タはコ－ドセグメント中に置かれるので、デ－タの最大領域は名前を省略しない場合に比べて小さくなります。
　省略時・・・コ－ド＋デ－タ≦６４Ｋ
　指定時・・・デ－タ≦６４Ｋ
　　コ－ド≦６４Ｋ

(2) レジスタ指定は、定義されたレジスタをこのセグメントのベ－ス・アドレスを持つレジスタとして使用するという意味しかなので、実行部分でベ－ス・アドレスを設定する必要があります。
例）

```
SEGMENT　(DATA)　　USING　DS
   ・
END
   ・
MODULE　　SAMPLE
          AX　=　DATA
          DS　=　AX　　あるいは　　%CONNECT(DATA)
           ・
           ・                          ※SHALのMACROを使用した場合
           ・
```

※　リンク情報は付録Ｆを参照して下さい。

(3) データの宣言

・ 外部データ宣言

```
EXTERNAL    型指定   変数名
```

他のモジュールで宣言された領域を参照する時指定します。
この変数名のオフセットは、ＥＮＴＲＹで指定されたセグメントのベースからのオフセットではなく、ＥＸＴＥＲＮＡＬが指定されたセグメントのベースからのオフセットであることに注意してください。

・ 内部データ宣言

```
[ENTRY]    型指定          変数名［(n)｜= 初期値］
```

ＥＮＴＲＹ指定は宣言された変数を、ほかのモジュールから参照可能にします。
ＥＮＴＲＹ指定がない場合は、ローカル変数となります。
データの型指定

ＢＹＴＥ	：	バイト定義
ＷＯＲＤ	：	ワード定義　（２バイト）
ＤＷＯＲＤ	：	２倍語定義　（４バイト）
ＱＷＯＲＤ	：	４倍語定義　（８バイト）
ＴＢＹＴＥ	：	１０バイト定義
レコード名	：	レコード構造定義
構造体名	：	構造体定義

（ｎ）　：　ｎ＝０　の場合は、実際の領域は確保されず、次の変数と同じ領域位置と見なされます。
ｎ≧１　の時は、配列の要素数を意味します。

初期値　：　変数に与える初期値を書きます。
書き方は次のように記述します。
・‘Ａ’,‘Ｂ’,‘Ｃ’ あるいは　‘ＡＢＣ’
・４１Ｈ，４２Ｈ，４３Ｈ　←　１６進で表示した場合
・４１４２４３Ｈ
・６５，６６，６７　←　１０進数で表示した場合

３．２　モジュール部

3.2.1 モジュールの記述は下記の通りです。

```
MODULE  モジュール名  ［(セグメント名)］［ＭＡＩＮ］［リンク情報］
      ・
      ・  （本　体）
      ・
END                （モジュール名）
```

セグメント　：モジュールを含むセグメント名を指定します。
省略時は@ＣＯＤＥになります。

ＭＡＩＮ　：ＭＡＩＮモジュールであることを記述し、このモジュールから実行します。

リンク情報　：付録Ｆを参照して下さい。

※　モジュールは、すべて他のモジュールからＣＡＬＬすることができます。

3.2.2 手続き

```
PROC            手続き名  [(FAR) | (NEAR)]
  ・
  ・
  ・
END             (手続き名)
```

手続きは、ＣＡＬＬ文によって参照することができます。

この手続きの属性には、ＮＥＡＲとＦＡＲがありますが、これはＤＥＣＬＡＲＥ部のＥＮＴＲＹ文で指定すれば、自動的に生成されます。

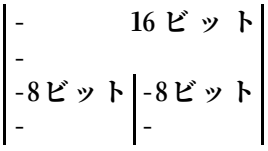
例）

```
DECLARE
   ENTRY    PROCEDURE    (FAR)    SAMPLE
END
  ・
  ・
PROC          SAMPLE
  ・
  ・
```

3.2.3 レジスタの使用方法

ＳＨＡＬでは、通常の高級言語の場合使うことができないレジスタを使用することができます。

以下に示す２０個のレジスタの名前をそのまま使うことにより、きめ細いプログラミングが可能です。



ＡＸ：	ＡＨ	ＡＬ	アキュムレータ・レジスタ	
ＢＸ：	ＢＨ	ＢＬ	ベース・レジスタ	
ＣＸ：	ＣＨ	ＣＬ	カウンタ・レジスタ	汎用
ＤＸ：	ＤＨ	ＤＬ	データ・レジスタ	レジスタ
	ＳＰ		スタック・レジスタ	
	ＢＰ		ベース・ポインタ	
	ＳＩ		ソース・インデックス	
	ＤＩ		ディスティネーション・インデックス	
	ＣＳ		コード・セグメント・レジスタ	
	ＤＳ		データ・セグメント・レジスタ	セグメント
	ＳＳ		スタック・セグメント・レジスタ	レジスタ
	ＥＳ		エクストラ・セグメント・レジスタ	

ＳＨＡＬでは、すべてのレジスタをユーザが管理しなければなりません。

ＳＨＡＬの関数やＣの関数では、すべてのレジスタが変更されると考えるべきです。

したがって、ＳＨＡＬの関数あるいはＣの関数などを、ＣＡＬＬする時は、必ず自分が使っているレジスタを保存する必要があります［ＰＵＳＨ，ＰＯＰを利用する］。

3.2.4　モジュールの中では、つぎの文を使用することができます。

アセンブリ文	mnemonic［operand］　　ユーザーが書くアセンブラ・ソース
演算文	OPERAND　［＝｜＋＝｜－＝｜≫｜≪］　OPERAND AX　　［＊＝｜／＝］　OPERAND OPERAND　［＋＋｜－－］
条件文	IF　（式） 　　文 ELSE　IF　（式） 　　文 ELSE 　　文 END　（IF） IF　（式）　BREAK　［(n)］
繰返し文	WHILE　（式） 　　文 END　　（WHILE） FOR　（［式］;［式］;［式］） 　　文 END　（FOR）
選択文	SWITCH　（レジスタ名） CASE　（定数式） 　　文 DEFAULT 　　文 END　（SWITCH）
補助制御文	BREAK　［(n)］ GOTO　ラベル RETURN CALL　手続き名

ラベルのついた文	ラベル　：　文
マクロ参照	%マクロ名　（［引数1,］...）
Cの関数	関数名　　（［引数1,］...）
空文	

複合文	演算文　［！演算文］...

（1）演算文

O代入文　（＝）

［メモリ｜レジスタ名］＝　［　メモリ｜レジスタ名｜定数式　］

右辺のデータ値を左辺の変数、あるいはレジスタに代入する命令です。
両辺の組合せは以下のものが許されます。

・レジスタ＝レジスタ

例）　ＡＸ＝ＢＸ
　　ＡＸ＝ＳＳ
　　ＳＳ＝ＢＸ

※　ただし、セグメントレジスタ＝セグメントレジスタは指定できません。
また、左辺にＣＳも指定できません。

・レジスタ＝メモリ

例）　ＡＸ＝ＡＢＣ　（ＡＢＣは変数名）
　　ＳＳ＝［ＢＸ＋ＳＩ＋４］

・汎用レジスタ＝定数式

例）　ＡＸ＝１２３４Ｈ

・メモリ＝レジスタ

例）　ＡＢＣ＝ＡＸ
　　［ＢＸ］＝ＳＳ

・メモリ＝定数式

例）　ＡＢＣ＝１２３４Ｈ

両辺は同じ大きさでなくてはなりません。

O加算命令　（＋＝）

［汎用レジスタ名｜メモリ］　＋＝　［　汎用レジスタ名｜メモリ｜数値］

左辺の内容に右辺の内容を加え、その結果を左辺に格納する命令です。
両辺ともメモリの組合わせは許されません。

ＡＡＡ　＋＝　［ＢＸ＋ＤＩ］　（禁止）

レジスタおよびメモリは、８ビットあるいは１６ビット、両方使用可能ですが、
両辺は同じ大きさでなくてはなりません。

例）　ＡＸ　＋＝　ＷＯＲＤ　ＰＴＲ　［ＢＸ＋ＤＩ］
　　ＷＯＲＤ　ＰＴＲ＋２［ＢＸ］　＋＝　ＣＸ

O減算命令　（－＝）

［汎用レジスタ名｜メモリ　］　－＝　［　汎用レジスタ名｜メモリ｜数値　］

左辺の内容から右辺の内容を減じ、結果を左辺に格納する命令です。
メモリから、メモリの内容を減じることはできません。

ＡＡＡ　－＝　ＢＢＢ　（禁止）

両辺は同じ大きさでなくてはなりません。

O乗算命令　（＊＝）

［ＡＬ｜ＡＸ　］　＊＝　［　汎用レジスタ名｜メモリ　］

ＡＬあるいは、ＡＸ　レジスタの内容に乗算をほどこして、この結果をＡＸあるいは、ＡＸ、ＤＸに格納します。

AL	×	8ビットのレジスタ又は、メモリ	→	AX

AX	×	１６ビットのレジスタ又は、メモリ	→	DX	AX

この命令（＊＝）では符合なし数値しか扱えませんので、符号付き数値を扱う場合には、ＩＭＵＬを使用してください。

Ｏ除算命令　（／＝）

［ ＡＬ｜ＡＸ ］　／＝　［ 汎用レジスタ名｜メモリ ］

ＡＸあるいは、ＤＸ，ＡＸの内容を右辺の内容で除算して、その結果をＡＨ，ＡＬあるいは、ＤＸ、ＡＸに格納します。

被除算		除算		商	余り
AX	÷	8ビットのレジスタ又は、メモリ	→	AL	AH

〈16ｂｉｔ　÷　8 ｂｉｔ〉

DX	AX	÷	１６ビットのレジスタ又は、メモリ	→	AX	DX

〈32ｂｉｔ ÷ 16ｂｉｔ〉

ＯＩＮＣＲＥＭＥＮＴ　と　ＤＥＣＲＥＭＥＮＴ命令

［ 汎用レジスタ名｜メモリ ］＋＋
［ 汎用レジスタ名｜メモリ ］－－

レジスタやメモリの内容を［＋１、－１］します。

ＯＳＨＩＦＴ命令

［ 汎用レジスタ名｜メモリ ］〔〉〉｜〈〈〕［１｜ＣＬ］

・ 〉〉は、右シフト命令です。
・ 〈〈は、左シフト命令です。
・〈〈は、〈〈の左側のレジスタまたは、メモリの内容を１（あるいはＣＬにある値）ビット分上位（ＭＳＢ）側に移動させ、最下位にあつたビット（ＬＳＢ）側には０が入れられます。

（例）

	MSB							LSB
	1	0	1	0	1	0	1	1

〈〈 命令

CF									
1	←	0	1	0	1	0	1	1	0 ← 0

・〉〉も同様になります。

（例）

	1	0	1	0	1	0	1	1	

〉〉 命令

									CF
0	0	1	0	1	0	1	1	0	→ 1

→

（２）ＩＦ文

```
［ラベル名：］　ＩＦ　　（条件式）
                 ：　　文１
 〃　　　｜ＥＬＳＥ△ＩＦ　（条件式）　　　｜
           ｜  ：　　文２　　　　　　　　｜
 〃　　　｜ＥＬＳＥ　　　　　　　　　　　｜
           ｜　　文３　　　　　　　　　　｜
 〃　　　ＥＮＤ　（ＩＦ）
```

条件式が満足されたとき、対応する文が実行されます。

```
［ラベル名：］　ＩＦ　　（条件式）
                  ・
                  ・　　文１
                  ・
                ＥＬＳＥ
                  ・
                  ・　　文２
                  ・
                ＥＮＤ　（ＩＦ）
```

条件式が満足されれば文１が、満足されなければ文２が実行されます。

```
［ラベル名：　］　ＩＦ　（条件式）　［BREAK [(n)] | GOTO　ラベル］
```

条件式が満足されたとき、ＢＲＥＡＫ文又はＧＯＴＯ文を実行します。
そうでなければ、つぎの文を実行します。

（３）ＷＨＩＬＥ文

```
［ラベル名：］　ＷＨＩＬＥ　（条件式）
                  ・　　文
                  ・
                ＥＮＤ　（ＷＨＩＬＥ）
```

条件式が真である限り文を実行します。

（４）ＦＯＲ文

```
［ラベル名：］　ＦＯＲ（[ 式１ ][ ！.. ]；[ 式２ ]；[ 式３ ]）
                  ・
                  ・
                  ・
                ＥＮＤ（ＦＯＲ）
```

式１　：　ＦＯＲ文（複合文）の先頭で、最初に実行する演算文（複合文）を指定します。
式２　：　ＦＯＲ文（繰り返し）の終りを判定する条件式を指定します。
　　この条件が真である限り、ＦＯＲ文はくり返されます。この判定は、繰り返しの始めに行なわれます。式２を省略したとき、条件は常に真と見なされます。
式３　：　ＦＯＲ文の繰り返しの後に実行される文を指定します。
　※　（　）内の三つの部分を、それぞれ省略することができます。特に式２を省略すると無限ループになります。

（5）ＳＷＩＴＣＨ文

```
［ラベル名：］　ＳＷＩＴＣＨ（レジスタ名）
                ＣＡＳＥ（ｅ１）
                        文１
                         ［BREAK(n)　｜　GOTO　ラベル］
          ｜　　ＣＡＳＥ（ｅ２）　｜
                        文２
          ｜　ＤＥＦＡＵＬＴ　　｜
                        文３　　｜
                ＥＮＤ　（ＳＷＩＴＣＨ）
```

ｅｉ（e1,e2…）：定数式、アセンブリ文（ＣＭＰレジスタ名ｅｉ）に変換されますので、アセンブラのＣＭＰ文の文法に合うように記述して下さい。

ＳＷＩＴＣＨ文のレジスタの値と、同一の値をｅｉに持つＣＡＳＥ文から実行され、ＢＲＥＡＫ文あるいはＧＯＴＯ文が来るまで、実行されたあと制御が他に移されます。

もし、文１の終りにＢＲＥＡＫ文あるいはＧＯＴＯ文等の記述がなく、制御が他に移されないときには、文２をつづいて実行します。

いずれのｅｉとも合わなかった場合には、ＤＥＦＡＵＬＴ指定があればその文（文３）を実行します。

（6）ＢＲＥＡＫ文

```
［ラベル名：　］　ＢＲＥＡＫ　［　（ｎ）　］
```

現在のネストから数えてｎ個分外側のＥＮＤ文に分岐します。（ｎ）が省略されたときには、（２）を補います。

注意　・ＩＦ（　）ＢＲＥＡＫ（１）はエラーになります。
　　・手続きの中からその外側に、ＢＲＥＡＫ文で制御を移すことはできません。

※ ネストの数は下記の文により増加されます。

ＷＨＩＬＥ
ＦＯＲ
ＳＷＩＴＣＨ
ＰＲＯＣ
ＩＦ

例）

```
nest 0          MODULE    SAMPLE
     1            PROC    SAM
                    :
     2              IF ( CX <>0 )
                       :
     3                WHILE ( BX == 0 )
     4                  IF ( CX == 1 )  BREAK ( 3 )
      3                  :
      4                  IF ( CX == 2 )  BREAK [ (2) ]
                         :
      3                END  ( WHILE )
                        :
      2              END  ( IF )
     1             END   ( SAM )
     0             END  ( SAMPLE )
```

（7）マクロの参照

```
%  マクロ名     ([引数1,]...)
```

最初が%であるときマクロの参照として認識されます。
マクロ名　：　＃ＭＡＣＲＯで定義された名前（3.4.3 参照）
引数　　：　＃ＭＡＣＲＯで定義されたパラメ－タで＃１～＃９に対応します。
・＃１～＃９までの仮変数は、引数の順序に対応します。

※　注意事項
・＃１～＃９を含むマクロを定義した場合、それを参照するときは、対応する引数を必ず指定しなければなりません。

（8）Cの関数呼び出し

```
関数名     (引数1,]...)
```

Cの関数はCと全く同じ表記法により呼び出すことができます。
引数にレジスタも使用できますが、ＡＸレジスタはＳＨＡＬが使用しますので注意して使用して下さい。
詳細は付録Cを参照して下さい。

３．３　条件式

条件式

```
(単純条件    [ & &単純条件  |  ||単純条件 ]...)
```

単純条件　：1.	変数名 レジスタ名 定数式	比較子	変数名 レジスタ名 定数式	
2.	%ＣＡ＿ＯＮ		carry flagが　ＯＮの場合	
	%ＳＩ＿ＯＮ		sign flagが　ＯＮの場合	
	%ＰＡ＿ＯＮ		parity flagが　ＯＮの場合	
	%ＯＶ＿ＯＮ		overflow flagが　ＯＮの場合	
	%ＣＡ＿ＯＦＦ		carry flag が　ＯＦＦの場合	
	%ＳＩ＿ＯＦＦ		sign flag が　ＯＦＦの場合	
	%ＰＡ＿ＯＦＦ		parity flag が　ＯＦＦの場合	
	%ＯＶ＿ＯＦＦ		overflow flagが　ＯＦＦの場合	

比較子　：　＝＝,　〈〉,　＞,　＞＝,　＞＞,　＞＞＝
〈,〈＝,　〈〈,　〈〈＝
条件として比較子のみも可となります。
単純条件の優先順位は　(),　＆＆,　||の順になります。
例）（ＡＬ＝＝' '　＆＆　%ＰＡ＿ＯＦＦ）の場合は、次のようにアセンブラに変換されます。

```
        CMP     AL, ' '
        JNE     ｌａｂｅｌ＿ｆａｌｓｅ
        JP      ｌａｂｅｌ＿ｆａｌｓｅ
          .
          .
          .
```

注意　・論理比較と算術比較について（ＰＬ／Ｇと反対なので注意）
ＳＨＡＬの条件式により、論理比較と算術比較のいずれも行うことができます。
論理比較は、１６ビットの場合、０　～　６５５３６　まで比較ができます。
(〉,　＞＝,　〈,　〈＝)
算術比較は、１６ビットの場合、－３２７６８　～　３２７６７まで比較ができます。
(〉〉,　＞＞＝,　〈〈,　〈〈＝)

３．４　制御指令

3.4.1　＃ＩＮＣＬＵＤＥ　宣言

```
＃ＩＮＣＬＵＤＥ　　ファイル名
```

ファイル名で指定されるファイルを読んで処理します。

注意：ファイル名にパス名をつけることもできますが、全体が１５文字以下でなければなりません。

3.4.2　＃ＤＥＦＩＮＥ　宣言

```
＃ＤＥＦＩＮＥ　　名前　　値
```

プログラム中で名前が使用された場合、名前を値に変換します。名前は大文字でも小文字でもかまいませんが、すべて大文字として認識します。

名前としてＳＨＡＬの予約語の使用は禁止されています。アセンブラの予約語についてはチェックしません。

名前として使用できる文字は、先頭が 'A' ～ 'Z' '0' ～ '9' 、 '@' 、'_' 、 '?' 、 '$' です。

ただし、最初の文字として '0' ～ '9' は用いることができません。

名前の長さは、１５文字までです。

3.4.3　#MACRO　定義

```
#MACRO　　マクロ名

　　マクロ本体
　　・
　　・
＃ＥＮＤ　　（マクロ名）
```

マクロ名を１５文字以内で定義します。この文の次の行から＃ＥＮＤ文の前の行までを、マクロとして登録します。

マクロ本体には、ＳＨＡＬ及びアセンブラの文が共に記述可能です。

各マクロに於て、パラメータを使用することができ、＃１から＃９までの９個が使用可能です。

注意　・　マクロ定義の総数は、１００個までです。
　　起動時ＭＡＣＲＯ．ＢＩＮから展開されます。
　　・　バラメータの記述において、＃と数字の間に空白があってはなりません。

例）
```
#MACRO　　CONNECT
　　MOV　　DX，＃1
　　MOV　　DS，DX
＃ＥＮＤ　　（ＣＯＮＮＥＣＴ）
使用法　　%CONNECT（０１００Ｈ）
```

3.4.4　＃ＩＦＤＥＦ，＃ＩＦＮＤＥＦ，＃ＥＬＳＥ，＃ＥＮＤＩＦ

```
＃ＩＦＤＥＦ　　名前１
　　文１
＃ＥＬＳＥ
　　文２
＃ＥＮＤＩＦ
＃ＩＦＮＤＥＦ　　名前２
　　文３
＃ＥＬＳＥ
　　文４
＃ＥＮＤＩＦ
```

＃ＩＦＤＥＦの場合は、名前１が＃ＤＥＦＩＮＥ文で定義されている場合文１を、されていない場合文２を処理します。

＃ＩＦＮＤＥＦは、逆に名前２が＃ＤＥＦＩＮＥ文で定義されていない場合文３を、されている場合文４を処理します。

```
例）
        ・
        ・
    #DEFINE    DEBUG    1
        ・
        ・
        ・
    #IFDEF     DEBUG
                文1
    #ELSE
                文2
    #ENDIF
```

この場合　、ＤＥＢＵＧが＃ＤＥＦＩＮＥで定義されているので、文１が処理され、文２は処理されません。

＃ＩＦＤＥＦと＃ＩＦＮＤＥＦは、最大１５の深さまでのネストができます。

３．５　データの参照について

次にプログラム中にデータを置き、そのデータをレジスタに転送する方法を説明します。

```
MODULE      SAMPLE

    SEGMENT  (DATA)  USING  DS
        BYTE  AAA      =    'A'
        WORD  BBB      =    'CD'
    END
              ・
              ・
              ・
            %CONNECT(DATA)

              AL  =  AAA
              BX  =  BBB
              CX  =  0FFH
            BBB  =  CX

END         (SAMPLE)
```

セグメントで宣言された変数名を、そのまま使うことにより初期値（Ｃあるいは、現在の変数の値）を利用することができます。

例）

```
  AL  =  AAA
```

領域ＡＡＡの、初期値４１ＨをＡＬに転送します。

```
  BX  =  BBB
```

これもＢＢＢの内容をＢＸに転送します。

```
  CX  =  0FFH
```

これは、レジスタＣＸに定数を入れます。また、レジスタのかわりに変数名を記述することにより、変数に直接定数を入れることもできます。

```
  AAA  = 'X'
  BBB  =  CX
```

ＢＢＢという変数に、レジスタＣＸの値を入れます。

※　＝の左右ともに、変数名を書くことはできませんので注意してください。

例）　　（誤）　ＡＡＡ　＝　ＢＢＢ

（正）　ＡＸ　＝　ＢＢＢ
　　ＡＡＡ　＝　ＡＸ

ＡＡＡ　と　ＢＢＢ　が変数名の場合

次に、レジスタを使わないで、定数を直接変数に代入する方法を説明します。
まず、転送するデータのサイズを決めなければなりませんので、次のように定義します。

（１）ＳＥＧＭＥＮＴで定義された型指定をそのまま用いる場合

```
ＳＥＧＭＥＮＴ　（ＤＡＴＡ）
　　・
　　・
ＢＹＴＥ　　ＳＡＭＰＬＥ　（５）
　　・
　　・
ＥＮＤ
　　・
　　・
ＳＡＭＰＬＥ　＝　'２'
```

（２）型指定の定義と異なる場合、代入文で型を明確にする。

```
ＳＥＧＭＥＮＴ　（ＤＡＴＡ）
　　・
　　・
ＷＯＲＤ　　ＸＸＸ
　　・
　　・
ＥＮＤ
　　ＢＹＴＥ　ＰＴＲ　ＸＸＸ　＝　'２'
```

（３）型指定の定義が無い場合も代入文で型を明確にする。

```
ＢＹＴＥ　ＰＴＲ　［ＢＸ＋ＳＩ］　＝　'２'
ＷＯＲＤ　ＰＴＲ　［ＢＰ＋ＤＩ］　＝　'ＡＢ'
```

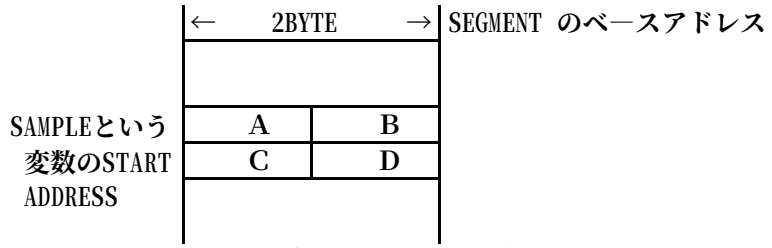
上記のように、ＷＯＲＤ　ＰＴＲ，ＢＹＴＥ　ＰＴＲ　あるいは、ＤＷＯＲＤ　ＰＴＲ　などの擬似命令を使用して、データの型を明確にしなければなりません。

３．６　アドレス指定について

ここでは、レジスタを用いた間接アドレス指定法について説明します。
間接アドレス指定法とは変数の持つ値を参照するのではなく、変数が持つ値をデータが置かれているアドレスと見なして参照を指定する方法です。

例）

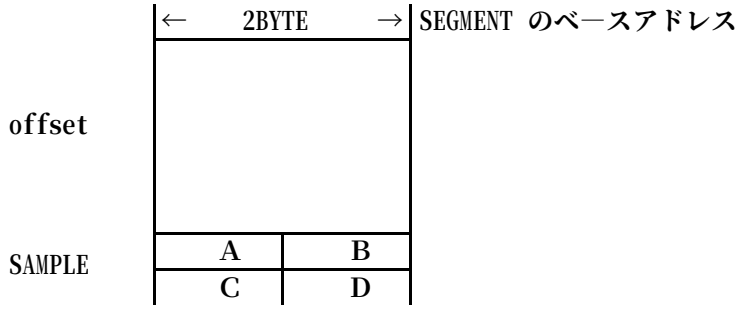
```
ＳＥＧＭＥＮＴ　（ＤＡＴＡ）
　　・
　　・
　　・
ＷＯＲＤ　　ＳＡＭＰＬＥ　＝　'ＡＢＣＤ'
　　・
　　・
ＥＮＤ
```



上記の例では、ＳＡＭＰＬＥという変数名を用いた場合は、ＡＢというデータが利用できます。それに対して変数のあるアドレスを利用して、間接指定ができますが、この時にはまず、ＯＦＦＳＥＴという擬似命令を用いてアドレスを取り出します。

ＢＸ　＝　ＯＦＦＳＥＴ　ＳＡＭＰＬＥ

上記のように書けば、ＳＥＧＭＥＮＴのベ－スアドレスからＳＡＭＰＬＥという変数が置かれたところまでの距離（これをオフセットといいます）が、ＢＸレジスタに入ります。



もし、ＳＥＧＭＥＮＴのベ－スアドレスからＳＡＭＰＬＥまでの距離が、００３ＤＨならばＢＸには、'００３Ｄ'が入ります。

さて、このオフセットがわかっている場合は、次のように［　］を用いた間接指定ができます。

ＡＸ　＝　［００３ＤＨ］

003DH
A B C D

この例ではＡＸに'ＡＢ'の値が転送されます。

この［　］の中に、アドレスを指定します。この方法で使用できる組み合わせは次の通りです。

［ＢＸ｜ＢＰ］＋［ＳＩ｜ＤＩ］＋［１６ビットの数値｜８ビットの数値］

それぞれの［　］から１個ずつ取り出した組み合わせです。また、ここにおける数値は直接の数値または、ＣＯＮＳＴ文によって定義されている数値に限ります。

次の方法は使用できませんので注意してください。

［　］のみ

［８ビット数値］

［ＢＰ］

使用例）

［ＢＸ　＋　ＳＩ　＋　１］　＝　ＡＸ

ＤＬ　＝　［ＢＰ　＋　ＤＩ］

ＤＸ　＝　－３［ＢＰ］

４．エラー　メッセージ

ERROR CODE	CONTENTS
５１	INVALID　KEYWORD　AFTER　　'#' OR '／'
	・ '#' , '／' のあとにKEYWORD がこなかった /DECLARE,/SEGMENT,/MODULE,/END. # INCLUDE,#MACRO,#END.
５２	CAN'T　USE　RESERVED　WORDS
	・SWITCH文CASE指定の（）の中に比較子やFLAGの名前（%PA__ON…）が指定された.
５３	NEED　SOME　KEYWORD
	・構造体のEND 文の（）の中にKEYWORD がなかった
５４	MISSING　OPERAND
	・FNSEGMENTのENTRY文の中でDATEの名前とか '（' あるいは初期値を 定義する時 '＝' がなかった. ・FOR文の（）の中で演算文の指定がなかった. ・IF文の（）の部分がなかった.
５５	MISSING '）'
５６	ILLEAGAL　EXPRESSION　　TYPE
	・条件式（IF文とかWHILE 文の条件式）のSYNTAX が間違っていた。 例）　%PA__ON　◇ ○　　（×） 　　CX＆＆　１　　（×） ・　＝, ＋＝, －＝・・・の文のSYNTAXが間違つた.
５７	MISSING　　'（'
５８	NEED THE RIGHT PARENTHESIS
	・ '（' が重複された.
５９	NEED NUMERICAL EXPRESSION
	・break 文の（）の中で数字がなかった場合.
６０	CAN NOT　FIND　FILE　NAME
	・#INCLUDE　文でファイル名がなかった場合
６１	NEED REGISTER　NAME
	・GROUP 文とかSEGMENT 文のUSING あとで CS, DS, SS, ES がなかつた場合.
６２	BAD　SYNTAX　IN　C FUNCTION
	・C の関数を定義する時、引数の定義が間違った場合
６３	OVER　MAXIMUM　CHARCTERS
６４	CAN'T　USE KEYWORD OF　SHAL　IN　DEFINE
	・USERがSHALのKEYWORD をDEFINE文で名前として使用した場合
６５	EXPECT CONTENTS
	・#MACROとか #DEFINE文の内容がなかった場合.
６６	EXPECT LABEL OR　SYMBOL　OR　NAME
６７	MISSING　'＝' ・CONST 文で '＝' がなかつた場合
６８	NEED RIGHT KEYWORD ・RECORD文で BITがなかった場合 ・ENTRY 文とか　EXTERNAL文でLABEL,PROCEDURE がなかった場合

ERRORE CODE	CONTENTS
６９	ILLEGAL SEGMENT DEFINE WITH OPTION －C
	-C OPTION を使う時には QDATAB, QDATAC, QDATAV, QDATAT,QDATAU,QDATAIの以外のSEGMENT 名は許されない.
７０	NEED OPTION -C TO USE C FUNCTION
７１	ILLEGAL GROUP DEFINE WITH OPTION -C
	-C option を使う時 DGROUP という CI コンパイラが管理させるDATA SEGMENTを定義しません.
７２	NEED ALPHABET
７３	CAN USE ONLY BREAK OR GOTO STATEMENT
７４	EXPECT SOME SEGMENT RAGISTER
７５	MACRO IS NOT REGISTERED.
８０	ILLEGAL BLOCK /STRUCTURE
	・/ DECLARE, /SEGMENT,/MODULE などBLOCK の構造が間違った場合.
８１	NEST OVERFLOW
	・NESTが３０以上の場合
８２	BREAK REFER UNDERFINED STRUCTURE
	・BREAK 文で定義されたBLOCK 構造がなかつた. （nestが ０ 以下になった場合）
８３	MISSING IF STATEMENT
	・ELSEがくる前にIF文がなかった場合
８４	MISSING SWITCH OR CASE STATEMENT
	・CASEあるいはDEFAULT 文がくる前にSWITCHあるいはCASE文がなかった
８５	MISMATCH BLOCK NAME
	・END 文がこのBLOCK のHEADとあわなかった （END 文の（）の中で書く名前とBLOCK の名前と間違った場合）
８６	NESTING CONFLICT
	・BLOCK の構造が正常でない場合 これは全体のBLOCK の構造中 どこかに間違い部分があれば、エラー が出ます.
８７	TOO MANY ENTRY PROCEDURE
	・／DECLARE で定義された ENTRY PROCEDURE・・が１００個以上の場合
８８	TOO MANY MACRO IS DEFINED
	・定義された MACROが１００個以上の場合
８９	INSUFFICIENT MEMOLY
９０	TOO MANY MACRO ARGUMENT
	・MACRO の argumentが９個以上の場合
９１	EXPEXT END STATEMENT
	・record文の最後がEND 文ではなかった場合
９２	OVERFLOW MACRO LEVEL
	・#MACROの中で MACROを利用する時このlevel が３つ以上の場合

ERRORE CODE	CONTENTS
９３	MISMATCH ARGUMENT IN MACRO
	・定義された MACROのargumentの個数と / MODULE 部分で書いた MACROのargumentの個数が間違った場合
９４	TOO MANY DEFINE STATEMENT
	・#DEFINE 文が１００個の以上の場合
９５	LABEL IS DUPLICATED
	・ENTRY PROCADURE 文で定義された名前の中で同一の名前が発見された場合
９６	MACRO IS DUPLICATED
	・定義された MACROの名前で再定義する場合
９７	DEFINE IS DUPPLICATED
	・定義された DEFINEの名前で再定義する場合
９８	ILLEGAL OPTION ARGUMENT
	・オプション指定の誤り

付録Ａ：ＳＨＡＬの制限

１．複合ブロックの入れ子の最大数　３０
２．宣言可能なＭＡＣＲＯの最大数　１００
３．宣言可能なＤＥＦＩＮＥの最大数　１００
４．ＭＡＣＲＯで指定可能な引数の個数　９
５．ＭＡＣＲＯの入れ子の数　３
６．名前として使用可能な文字
'A'～'Z' ，'0'～'9' ,'@' ，'?' ，'_' ，'$'
７．使用できないラベル名文字　＿１００００～＿１９９９９
８．名前の有効文字長　４０文字
ただし、ＤＥＦＩＮＥで定義する名前は１５文字まで
９．ＦＩＬＥ名の有効文字長　３２文字

付録Ｂ：ＳＨＡＬの変換規則

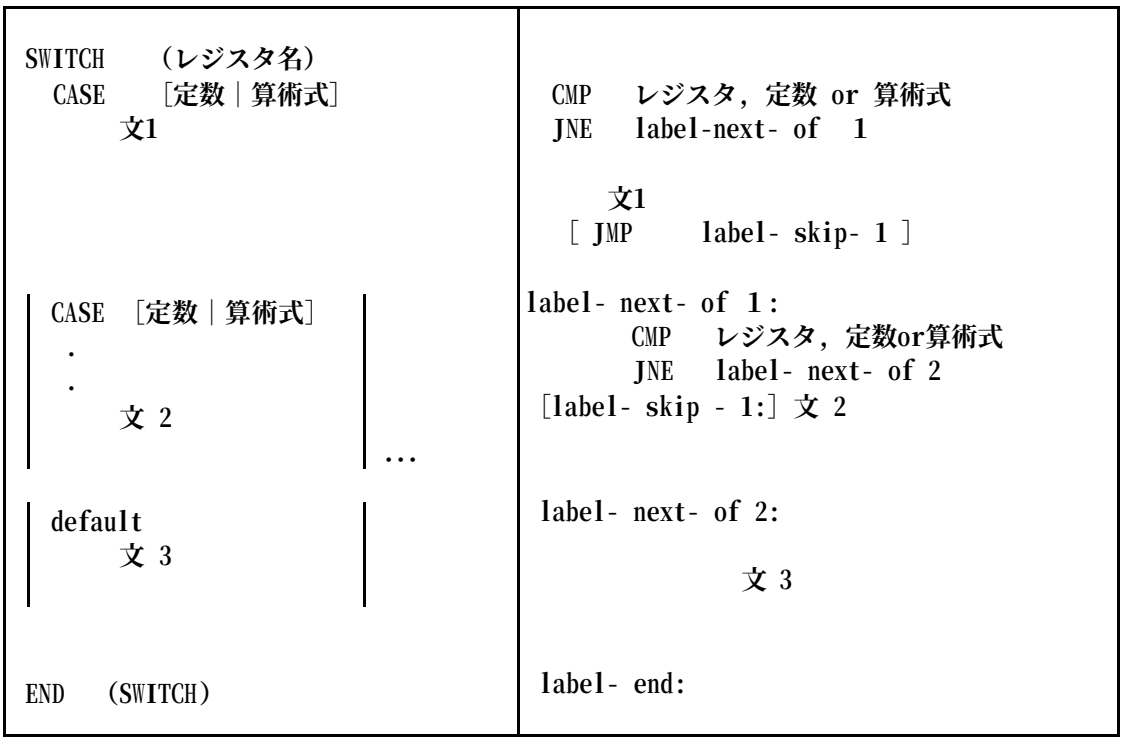
SHAL の ソース	ＳＨＡＬの出力コード
空文	無視
;文字列	;文字列
／文字列	文字列（先頭の／を削除し，文字列を左につめる）
# INCLUDE ファイル名	入力ファイルの 変更
# DEFINE 名前 文字列.	なし
# MACRO 名前 ・ ・ ・ # END （名前）	なし
DECLARE ・ ・ ・ END	なし
CONST 定数名 ＝ 定数式	定数名 EQU 定数式
RECORD 名前 フィールド名BIT bit幅[VALUE定数式] ・ ・ END （名前）	名前RECORDフィールド名：幅［＝式］..

STRUCT　構造体名 . . . END　（構造体名）	構造体名　STRUC . . . 構造体名　ENDS
GROUP グループ名［USING seg-reg ］ segment 名［, segment 名］... END　（グループ名）	グループ名　GROUP segment名, ... ［ASSUME　seg.reg:グループ名］

SHAL の　ソース	ＳＨＡＬの出力コード
EXTERNAL LABEL［(far) ｜(near)］ ラベル名	EXTRN　ラベル名 ：［far ｜near］
ENTRY　LABEL　ラベル名	PUBLIC　ラベル名
EXTERNAL PROCEDURE［(far) ｜(near)］ 手続き名	EXTRN　手続き名 ：［far ｜near］
ENTRY　PROCEDURE　手続き名	PUBLIC　手続き名
SEGMENT （セグメント名） .　［USINGseg-reg］［文字列］ . . . END　（セグメント名）	セグメント名SEGMENT ［文字列］ ［ASSUME　seg-reg:セグメント名］ . . . セグメント名　ENDS
EXTERNAL　型指定　変数名	EXTRN　変数名　：型指定
［ENTRY ］　型指定 レコード名 構造体名　名前　（０）	［PUBLIC　名前］ 名前　LABEL　型指定 レコード名 構造体名
［ENTRY ］　型指定　名前　（ｎ）	［PUBLIC　名前］ 名前　DB　n　DUP　（？） . . DT
［ENTRY ］　型指定　名前　＝　初期値	［PUBLIC　名前］ 名前　DB　初期値 . . DT
［ENTRY ］　型指定　名前	［PUBLIC　名前］ 名前　DB　？ . . DT

MODULE　モジュール名［MAIN］［文字列 ］ . . . . END　　（モジュール名）	@CODE　SEGMENT　［文字列］ ASSUME　CS:@CODE ［？モジュール名：］ モジュール名　PROC　FAR . . . . モジュール名　ENDP @CODE　ENDS　［？モジュール名］

SHAL の　ソース	ＳＨＡＬの出力コード
PROC　手続き名 ・ ・ ・ ・ END　（手続き名）	名前　PROC　［FAR ｜NEAR］ ＊ FAR／NEAR は ／DECLARE の中での ENTRY PROCEDURE ［(FAR) ｜(NEAR)］名前の FARあるいはNEARを使います。 名前　ENDP
IF　（条件式） . . 文　true . ELSE . . . .　文false END　(IF)	条件式の処理 JMP 偽　Lfalse 文　true JMP　Label-end Lfalse: 文 false Label- end:
WHILE　（条件式） . .　文　true . END　(WHILE)	条件式の処理 JMP 偽　Lfalse 文 true Label-end:
FOR （式１；式２；　式３；） . . . 文 END　(FOR　)	式１　処理 Label- start : 式２　処理 JMP 偽　Label 文 式３　処理 JMP　Label- start Label:


SWITCH (レジスタ名)
CASE [定数｜算術式]
文1
CASE [定数｜算術式]
文 2
...
default
文 3
END (SWITCH)
CMP レジスタ, 定数 or 算術式
JNE label-next- of 1
文1
[JMP label- skip- 1]
label- next- of 1:
CMP レジスタ, 定数or算術式
JNE label- next- of 2
[label- skip - 1:] 文 2
label- next- of 2:
文 3
label- end:

SHAL の　ソース	ＳＨＡＬの出力コード
RETURN	RET
BREAK [(n)]	JMP　　ラベル
GOTO　ラベル	JMP　　ラベル
ａ　＝　ｂ	ＭＯＶ　　ａ，ｂ
ａ　＋＝　ｂ	ＡＤＤ　　ａ，ｂ
ａ　－＝　ｂ	ＳＵＢ　　ａ，ｂ
ａ　＊＝　ｂ	ＭＵＬ　　ｂ
ａ　／＝　ｂ	ＤＩＶ　　ｂ
ａ＋＋	ＩＮＣ　　ａ
ａ－－	ＤＥＣ　　ａ
ａ　〉〉ｎ	ＳＨＲ　　ａ，ｎ
ａ　〈〈ｎ	ＳＨＬ　　ａ，ｎ
IF　（条件式１） 　　文　1 ELSE　IF　（条件式２） 　　文　2 ELSE　　IF　（条件式３） 　　文　3 ELSE 　　文　4 END　（IF）	条件式１　　処理 　JMP 偽　　Label- next1 　　文１ 　　JMP　　Label- end Label- next1: 条件式２　　処理 　JMP 偽　　Label- next2 　　文２ 　　JMP　　Label- end Label- next2 　　. 　　. 　　. 　　. Label- end:
IF　（条件式）［BREAK[(N)]｜ 　　GOTO　ラベル］	条件式　　処理 jmp真　飛び先
%マクロ名　（引数１、・・・）	（マクロを展開）
名前　：　#DEFINE で　宣言した場合	(#DEFINE で宣言した CONTENTS に変換)
名前　　（引数１、・・・）	（ＣのインターフェイスによりＣＡＬＬ）

付録Ｃ：Ｃの関数の呼び方

ＳＨＡＬではＣ言語の関数を直接呼ぶことが可能です。すなわち、Ｃの標準関数やＣで記述された関数をＳＨＡＬで呼んで利用したり、逆にＳＨＡＬで記述した関数をＣで呼んで利用したりできます。
まず、Ｃの関数をＳＨＡＬで参照する方法を説明します。

Ｃ．１　Ｃの関数の参照法

Ｃ.1.1　ＳＨＡＬの起動で－Ｃ　ｏｐｔｉｏｎを指定します。
これは、Ｃ１－８６　Ｃｏｍｐｉｌｅｒが管理するＤＡＴＡ　ＳＥＧＭＥＮＴを定義し、ＳＨＡＬで記述するＤＡＴＡ　ＳＥＧＭＥＮＴの名前がＣＩ－８６ｃｏｍｐｉｌｅｒが管理するＤＡＴＡ　ＳＥＧＭＥＮＴの名前とあうかをチェックします。
もし、－Ｃを定義することなく、Ｃの関数を参照した場合には、次のように警告エラーが出ます。
ＮＥＥＤ　ＯＰＴＩＯＮ　－Ｃ　ＴＯ　ＵＳＥ　ＦＵＮＣＴＩＯＮ　ＯＦ　Ｃ　ＬＡＮＧしかしながら、参照しているＣの関数がすべてＳＨＡＬで記述されている場合には、上記のエラーは無視してかまいません。

Ｃ.1.2　プログラム上での記述はＣと同じように書きます。

関数名　（　[引数１,]....）

（１）　関数名　：Ｃ１の標準関数　（まはたはＵＳＥＲが書いた関数名）

（注意）　Ｃ１のコンパイル時に　－１　ｏｐｔｉｏｎを指定しなかった場合は、８文字しか認識しませんので、９文字以上は記述してはなりません。

－１　ｏｐｔｉｏｎ　を使った場合は３１字まで記述可能です。

８文字

ＬＯＮＧ－ＮＡＭＥ１
書いたらＬＩＮＫする時エラーが出ます。
（但し、－１を使った場合は例外）

（２）　引数

・整数値　：　０～９の文字列

（例）　ｐｒｉｎｔｆ　（"%ｄ¥ｎ",１０）

・アドレス：　最初が '&' の場合アドレスとして認識します。

&		ＤＳ ＤＳ ＤＳ ＤＳ		：　メモリの参照

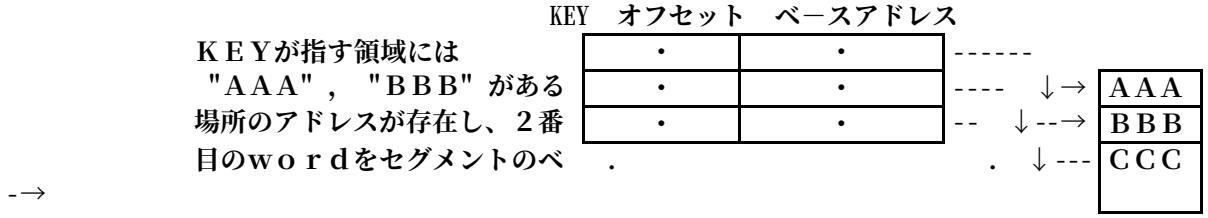
セグメント・レジスタとオフセットを組み合せてアドレスを求めます。セグメント・レジスタを省略した場合は、ＤＳレジスタがセグメント・レジスタとみなされます。

（例）　ｐｒｉｎｔｆ　（"%ｓ¥ｎ", &ＤＳ：ＳＡＭＰＬＥ）
ｐｒｉｎｔｆ　（"%ｓ¥ｎ", &ＳＡＭＰＬＥ）

ｐｒｉｎｔｆ　（"%ｓ¥ｎ", &ＳＳ：１２［ＢＰ］）

・ポインタ配列：　最初が '＊' の場合はポインター値として認識します。
これは、Ｃで次のように書いたポインタの配列を参照する時、使います。

＊ ＫＥＹ［ ］＝｛"ＡＡＡ"，"ＢＢＢ"，"ＣＣＣ"｝;



ースアドレスとして認識します。．←　DWORD　→　．

（例）　ｐｒｉｎｔｆ　（"%ｓ¥ｎ"，＊ＫＥＹ［SI］）

・レジスタ：　そのまま書きます。
（BX,CX,SI,DI,SP,BP,CS,DS,ES,SS ）が記述可能です。
ＡＸレジスタはＳＨＡＬが作業用に使用するため、引数とした時の動作は不定になります。

・文字列　：　"文字列"
・　" で囲まれた部分を文字列と見なします。
・　文字列の中では次の特殊指定を使うことができます。

・¥ａ　：　ｂｅｌｌ
・¥ｂ　：　ｂａｃｋ　ｓｐａｃｅ
・¥ｔ　：　ｔａｂ
・¥"　：　"
・¥'　：　¥'
・¥ｎ　：　ｃａｒｒｉｇｅ　ｒｅｔｕｒｎ
・¥ｖ　：　ｖｅｒｔｉｃａｌ　ｔａｂ
・¥¥　：　ｂａｃｋｓｌａｓｈ　ｃｈａｒａｃｔｅｒ
・¥ｏ　：　ｎｕｌｌ　ｃｈａｒａｃｔｅｒ
・¥ｆ　：　ｆｒｏｍ　ｆｅｅｄ

・　文字列の最後には　ｎｕｌｌ　ｃｈａｒａｃｔｅｒ（0：バイナリのゼロ）が付加されます。

・変数　：　上記以外の指定はすべて１６ビットの変数として認識します。

・　ｐｒｉｎｔｆ　（"¥ｔ　%　ｄ¥ｎ"，　ＶＡＲ）

（3）　リターン　コード　について

Ｃの関数はリターンコードを次のように返します。

・CHAR,SHORT,INTERGER はＡＸレジスタに値を返します。
・ＬＯＮＧの場合はＡＸとＤＸに返します。
ＡＸはオフセット
ＤＸはセグメントのベースアドレス
・ＤＯＵＢＬＥの場合はＡＸとＤＸ，ＢＸ，ＣＸに返します。

従ってＳＨＡＬで誕生した関数からの RETURN CODE も上記の規則にあうようにしなければなりません。

（４）　プログラム作成時の注意事項

・Ｃの関数では下記のレジスタが使われることがありますので、呼ぶ側で使用中のレジスタをＳＡＶＥする必要があります。

ＡＸ，ＢＸ，ＣＸ，ＤＸ，ＳＩ，ＤＩ

Ｃ．２　Ｃのインターフェースに合ったＳＨＡＬの関数の作り方

Ｃで参照可能な関数をＳＨＡＬで記述する上での注意点は次の通りです。

C.2.1　下記に示す標準エントリーロジックと標準リターンロジックを使います。

例）

```
                    。
                    。
                    。
                MODULE    SAMPLE      PUBLIC      BYTE   'CODE'
                          PUSH        BP
STANDARD  ENTRY  LOGIC    MOV         BP,SP       :    %SPINIT( )
                          。
                          。
                          。
                          。
                          。
                          MOV         SP,BP
STANDARD RETURN  LOGIC    POP         BP          :    %SP_END( )
              END         (SAMPLE)
```

※　ＳＨＡＬの標準ＭＡＣＲＯとして、ＳＰＩＮＩＴ，ＳＰ＿ＥＮＤは用意されています。

C.2.2　引数は現在のベースポインタより６バイト目が先頭位置になりますから、引数を＃ＤＥＦＩＮＥで定義して使えば見やすくなります。

例）

ＡＡＡ　（ＢＸ，"ＳＴＲＩＮＧ"，＆ＤＳ：ＡＤＤＲ）

上記の引数に対応して、下記のＤＥＦＩＮＥをする。

```
＃ＤＥＦＩＮＥ　　ＦＳＴ＿ＡＲＧ　　－６［ＢＰ］
＃ＤＥＦＩＮＥ　　ＳＮＤ＿ＡＲＧ　　－８［ＢＰ］
＃ＤＥＦＩＮＥ　　ＴＲＤ＿ＡＲＧ　－１２［ＢＰ］
    。
    。
    。
```

C．３　Cで記述したデータの参照の方法

Cで定義されたＤＡＴＡの参照方法を説明します。

Cの記述で

```
1      int     AAA;
2    CHAR      BBB;
3      int     CCC［10］;
4    CHAR      DDD［10］;
5      int     EEE［10］［5］;
6    CHAR      FFF［10］［5］;
7    CHAR   ＊GGG［ ］:{"AAA", "BBB", "CCC"};
```

のように定義されたものを ＳＨＡＬ のプログラムで参照する場合は下記のように指定します。

```
ＳＥＧＭＥＮＴ          @ＤＡＴＡＢ
   ＥＸＴＥＲＮＡＬ          ＷＯＲＤ     ＡＡＡ
   ＥＸＴＥＲＮＡＬ          ＢＹＴＥ     ＢＢＢ
   ＥＸＴＥＲＮＡＬ          ＷＯＲＤ     ＣＣＣ
   ＥＸＴＥＲＮＡＬ          ＢＹＴＥ     ＤＤＤ
   ＥＸＴＥＲＮＡＬ          ＷＯＲＤ     ＥＥＥ
   ＥＸＴＥＲＮＡＬ          ＢＹＴＥ     ＦＦＦ
   ＥＸＴＥＲＮＡＬ          ＤＷＯＲＤ   ＧＧＧ
ＥＮＤ          （@ＤＡＴＡＢ）
```

それぞれの参照は次の通りにします。

（１）　ＩＮＴＥＧＥＲ値
そのまま記述します。

```
ＡＸ＝ＡＡＡ
ｐｒｉｎｔｆ　（"%Ｃ￥ｎ", ＢＢＢ）
```

（２）　ＣＨＡＲＡＣＴＥＲ値そのまま記述します。

```
ＡＸ＝ＢＢＢ
ｐｒｉｎｔｆ　（"%Ｃ￥ｎ", ＢＢＢ）
```

（３）　ＩＮＴＥＧＥＲの配列
Cでの１づつの増減はＳＨＡＬでは２づつの増減に対応させます。
それは、ＩＮＴＥＧＥＲのＳＩＺＥがＷＯＲＤであるためです。

```
ＡＸ＝ＣＣＣ［ＳＩ］
ＡＸ＝ＣＣＣ＋２　　；ＣＣＣ［1］と同一
```

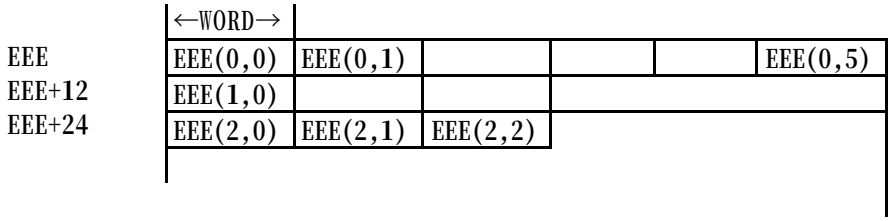
［ ］を使った場合は　ＲＥＧＩＳＴＥＲ の中で、ＳＩ、ＤＩ、ＢＸ、ＢＰしか、使用することができませんので注意してください。

（４）　ＣＨＡＲＡＣＴＥＲの配列
ＣＨＡＲの配列は、１ＢＹＴＥの長さですからCでの１つづつの増減はＳＨＡＬでも１つづつの増減になります。

```
例）　ＡＸ　＝　ＤＤＤ［ＳＩ］
　　ＡＸ　＝　ＤＤＤ＋１　　；ＤＤＤ［1］と同一
```

（５）　ＩＮＴＥＧＥＲの２次元配列

この場合は次のようになりますから１次元のｏｆｆｓｅｔに計算してａｄｄｒｅｓｓを求めなければなりません。



従って、ＥＥＥ［2］［2］は、ＥＥＥから２８ＢＹＴＥ目にあるものです。

例）　Ｃで

ＥＥＥ［2］［0］　＝　０；

という文を、ＳＨＡＬで書くと

ＳＩ　＝　１２＊２

ＢＹＴＥ　ＰＴＲ　ＥＥＥ［ＳＩ］　＝　0となります。

（６）　ＣＨＡＲの２次元の配列

ＩＮＴＥＲＧＥＲの場合とほぼ同一です。

例）　前頁の例を見ると

Ｃでは

ＥＥＥ［3］［2］＝０；

のものを、ＳＨＡＬでは

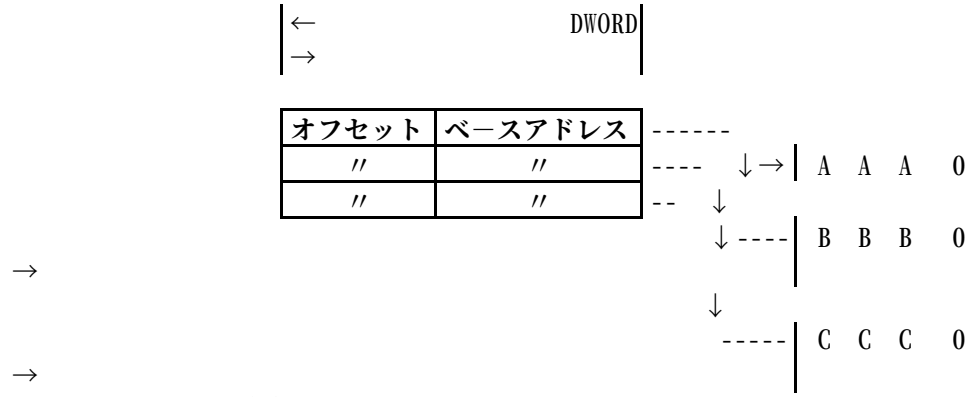
ＳＩ＝６＊３

ＳＩ＋＝２

ＥＥＥ［ＳＩ］＝0になります。

（７）　ＰＯＩＮＴＥＲの配列

これは、データの格納場所のポインタが配列として構成されたもので次のようになります。



例）　Ｃで

ｐｒｉｎｔｆ（"%Ｓ¥ｎ"，ＧＧＧ［1］）；

が、ＳＨＡＬでの記述では

ＳＩ　＝　１＊４

ｐｒｉｎｔｆ（"%Ｓ¥ｎ"，＊ＧＧＧ［ＳＩ］）；

前の例で、Ｃで定義したＤＡＴＡを間接指定法で参照したい時には、次のようにします。

まず、ＳＥＧＭＥＮＴの名前はすべて、@ＤＡＴＡＢとします。

ＳＥＧＭＥＮＴの中で、宣言されたデータを次のように指定します。

例）

```
SEGMENT          (a) DATAB
   EXTERNAL              WORD    SAMPLE
END              (a) DATAB

MODULE           ?AAA
        ・
        ・
        AX      =  OFFSET   DGROUP：SAMPLE
        ・
        ・
```

※　DGROUPとは、CIが作成するGROUP名で、Cで記述したデータは、すべて、DGROUPからのオフセットになりますので上記のように書けます。

上記の例の結果は、SAMPLEというデータのDGROUPからのオフセットが、AXに入れられます。

C．4　AUTOMATIC　変数の作成と使い方

AUTOMATIC変数は、現在のSP（スタック・ポインタ）の下方のMEMORYを使用します。

例えば

Cで

```
subroutin()
{
        int     A;
        char    C;
          ・
          ・
```

と記述したものは、SHALでは下記のように、DEFINEで指定します。

```
MODULE          SUBROUTI
      ・
      ・
#DEFINE      A?      WORD  PTR   －2［BP］
#DEFINE      C?      WORD  PTR   －4［BP］
```

上記のように、#DEFINE文で定義して、その名前を変数名として使用します。

```
AX   =  A?
C?   =  BX
```

AUTOMATIC変数は、STACK SEGMENT内に領域が確保されますから、間接指定の場合には、SSがベースアドレスを持つので

printf（"%S¥n"　&SS：A?）と記述します。

付録D：8086命令セット

ASM－86命令セット

D．1　概要

オペランド・タイプ記号

記号	オペランド・タイプ
numb	すべての数値式
numb8	8 ビットに評価される数値式.
acc	アキュムレータ．ＡＸ又はＡＬ.
reg	汎用レジスタ．セグメント．レジスタを除く.
reg16	16ビットの汎用レジスタ．セグメント・レジスタを除く.
segreg	セグメント・レジスタ．ＣＳ．ＤＳ．ＳＳ．ＥＳのいずれか.
mem	アドレス式．ベース・アドレッシング・モードまたはインデックス・アドレッシング・モード. あるいはその双方が含まれてもよい. ［例］ variable variable + 3 variable[bx] variable[SI] variable[BX + SI] [BX] [BP + DI]
simpmem	アドレス式（ベース・アドレッシング・モードとインデックス・アドレッシング・モードを含みません）. variable variable + 4
mem ｜reg	"mem" でも "reg" でもよい.
mem ｜reg16	"mem" でも "reg16" でもよい.
label	ラベルを表わすアドレス式.
lab8	"label" に同じ（但し現在のアドレスから±128 バイトの範囲内に限られます）.

８０８６のＣＰＵは１ビットのフラグ・レジスタを９個持っており、それがＣＰＵの状態を反映しています。ユーザは直接このフラグ・レジスタをアクセスできませんが、それを見て実行された命令がレジスタやオペランドに及ぼした効果について知ることができます。

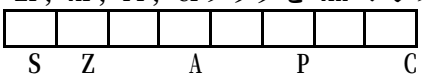
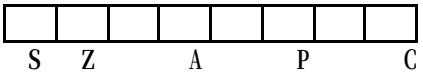
命令がフラグ・レジスタに与える効果は、次の表に示す９つのフラグ・レジスタ記号を用いて、命令定義表に記述されています。

フラグ・レジスタ記号

ＡＦ	補助キャリア・フラグ
ＣＦ	キャリ・フラグ
ＤＦ	ディレクション・フラグ
ＩＦ	インタラプト・イネーブル・フラグ
ＯＦ	オーバーフロ・フラグ
ＰＦ	パリティ・フラグ
ＳＦ	符号フラグ
ＴＦ	トラップ・フラグ
ＺＦ	ゼロ・フラグ

D．２　データ転送命令

データ転送命令

文法	実　行　結　果
IN acc,numb8｜numb16	命令の２番目のバイトで示めれる I/Oポートから、AL（８ビット操作）あるいは AX （１６ビット操作）のレジスタに入力する.
IN acc, DX	DXレジスタで示される I/Oポートから、AX（１６ビット操作）のレジスタに入力する.
LAHF	SF, ZF, AF, PF, CFフラグを AH レジスタへ転送します. 
LDS reg16, men	メモリ・アドレス（DWORD 変数）のセグメント値の部分をDSレジスタに送り、オフセット値の部分を１６ビット汎用レジスタに送ります.
LEA reg16, men	メモリ・アドレスのオフセット値を、１６ビットのレジスタへ転送します.
LES reg16, mem	メモリ・アドレスのセグメント値をESレジスタへ、オフセット値を１６ビット汎用レジスタへ転送します.
MOV reg, mem｜reg	メモリ又はレジスタを、セグメント・レジスタへ転送します.
MOV mem ｜reg, reg	レジスタを、メモリ又はレジスタへ転送します.
MOV mem ｜reg, numb	即値データを、メモリ又はレジスタへ転送します.
MOV segreg, memreg16	メモリ又はレジスタを、セグメント・レジスタへ転送します.
MOV mem ｜reg16, segreg	セグメント・レジスタを、メモリ又はレジスタへ転送します.
OUT numb8 ｜numb16, acc	命令２番目のバイトで示される I/Oポートに、AL（８ビット操作）あるいはAX（１６ビット操作）のレジスタの内容を出力する.
OUT DX. acc	DXレジスタで示される I/Oポートに、AL（８ビット操作）あるいはAX（１６ビット操作）のレジスタの内容を出力する.
POP mem ｜reg16	スタックの１番上の要素をメモリ又はレジスタへ転送します。
POP segreg	スタックの１番上の要素をセグメント・レジスタへ転送します。 ただし、CS（コード・セグメント）レジスタへの転送は許されません。
POPF	スタックの１番上の要素をフラグ・レジスタへ転送します。
PUSH mem｜reg16	メモリ又はレジスタを、スタックの１番上にプッシュします。
PUSH sereg	セグメント・レジスタの内容をスタックへプッシュします。
PUSHF	フラグをスタックへプッシュします。
SAHF	AHレジスタの内容をフラグへ転送します。 
XCHG reg, mem ｜reg	レジスタと、メモリ又はレジスタの内容を、入れ替えます。
XCHG mem｜reg, reg	メモリ又はレジスタとレジスタの内容を、入れ替えます。
XLAT mem｜reg	メモリ又はレジスタで与えられるテーブルのサーチを開始します。
	レジスタの場合はBXです。ALをBXからのオフセットALでおき替えます。
XLAT	(AL)-((AL)+(BX)) ALとBXの和で示されるデータ・バイトをALレジスタにロードする。

D．３　演算、論理、シフト命令

８０８６は四則演算をいろいろな方法で実行します。演算は、８ビットでも１６ビットでもよい。

９個の１ビット・フラグのうち６個は、たいていの算術演算によって（その演算の結果いかんに従って）セット／クリアされます。

次の表に、演算命令のフラグへ及す効果をまとめました。演算命令、論理命令とシフト命令それぞれの定義表を順に示します。

フラグへ及ぼす効果

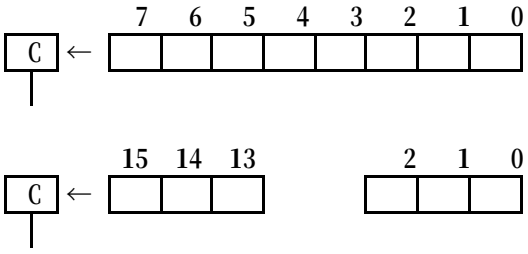
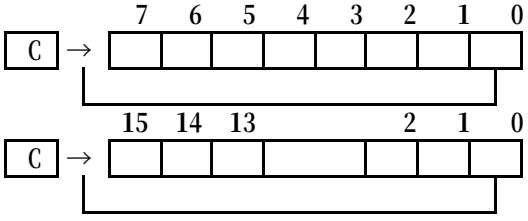
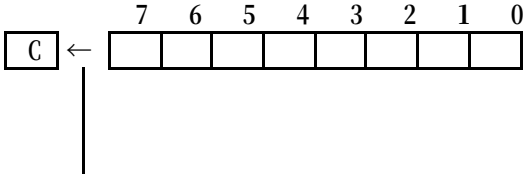
ＣＦ	演算効果が、上位ビットにけた上り（加算時）又は上位ビットよりけた下り（減算時）するときにセットされます。そうでないときは、クリアされます
ＡＦ	演算結果が、下位４ビットにけた上り（加算時）又はけた下り（減算時）するときにセットされます。そうでないときは、クリアされます。
ＺＦ	演算結果がゼロのときにセットされ、ゼロでないときにはクリアされます。
ＳＦ	演算結果が負のときにセットされます。
ＰＦ	演算結果の下位の８ビットが、偶数パリティならばセツトされ、奇数パリティならばクリアされます。
ＯＦ	演算結果がオーバーフロ（結果のサイズが受け手側のサイズより大きい）のときにセットされます。

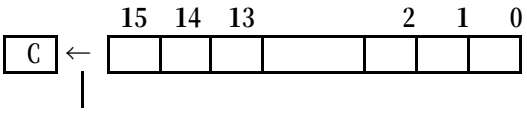
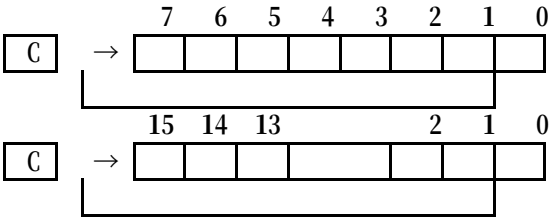
演算命令

文法	実　行　結　果
AAA	加算に際し、アンパック BCD(ASCII) を調整します。ALが調整されます。
AAD	除算に際し、アンパック BCD(ASCII) を調整します。ALが調整されます。
AAM	乗算に際し、アンパック BCD(ASCII) を調整します。AXが調整されます。
AAS	減算に際し、アンパック BCD(ASCII) を調整します。ALが調整されます。
ADC reg，reg｜men	メモリ又はレジスタの内容をレジスタに加えます（キャリ付き)。
ADC mem ｜reg，reg	レジスタの内容をメモリ又はレジスタに加えます（キャリ付き)。
ADC mem ｜reg，numb	即値をメモリ又はレジスタに加えます（キャリ付き)。
ADD reg，mem｜reg	メモリ又はレジスタの内容をレジスタに加えます。
ADD mem ｜reg，reg	レジスタの内容をメモリ又はレジスタに加えます。
ADD mem ｜reg，numb	即値をメモリ又はレジスタに加えます。
CBW	ALバイト符号拡張してAXワードに拡張します。
CWD	AXI ワードを符号拡張して、AX/DX2ワードに拡張します。
CMP reg，mem｜reg	メモリ又はレジスタと、レジスタを比較します。
CMP mem ｜reg，reg	レジスタと、メモリ又はレジスタを比較します。
CMP mem ｜reg,numb	定数とメモリ又はレジスタを比較します。
DAA	１０進加算結果を調整します。ALが調整されます。
DAS	１０進減算結果を調整します。ALが調整されます。
DEC mem ｜reg	メモリ又はレジスタから１を引きます。
INC mem ｜reg	メモリ又はレジスタに１を足します。
DIV mem ｜reg	アキュムレータAX又はALを、メモリ又はレジスタ（符合なし）で割ります。商がバイトのときにはALを商、AHに余りが入り、ワードのときはAXに商、DXに余りが入ります。
IDIV mem｜reg	アキュムレータAX又はALを、メモリ又はレジスタ（符号付きで）割ります。商と余りは DIVに同じ。

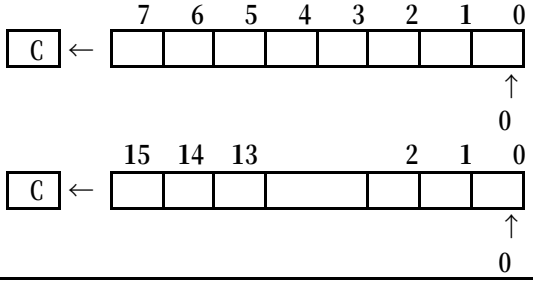
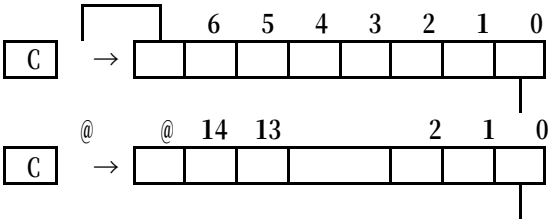
IMUL mem｜reg	メモリ又はレジスタにアキュムレータAX又はALを（符合付きで）かけます。積はバイト演算のときAH,AL に入り、ワード演算のときDX,AXに入ります。
MUL mem ｜reg	メモリ又はレジスタにアキュムレータAX又はALを（符号なしで）かけます。積はIMULに同じ。
NEG mem ｜reg	メモリ又はレジスタの２つの補数をとります。
SBB reg，mem｜reg	メモリ又はレジスタの内容をレジスタから引きます（ボロー付き）
SBB reg，mem｜reg	メモリ又はレジスタの内容をレジスタから引きます（ボロー付き）
SBB mem ｜reg，reg	レジスタの内容を、メモリ又はレジスタから引きます(ボロー付き)
SBB mem ｜reg，numb	即値をメモリ又はレジスタから引きます(ボロー付き)。
SUB reg，mem｜reg	メモリ又はレジスタの内容をレジスタから引きます。
SUB mem ｜reg，reg	レジスタの内容を、メモリ又はレジスタから引きます。
SUB mem ｜reg，numb	即値をメモリ又はレジスタから引きます。

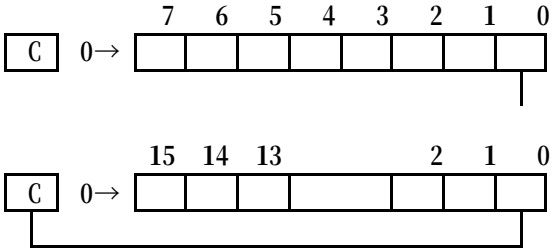
論理命令とシフト命令

文法	実　行　結　果
AND reg，mem｜reg	レジスタと、メモリ又はレジスタのビット毎の論理積をとります。
AND mem ｜reg，reg	メモリ又はレジスタとレジスタとのビット毎の論理積をとります。
AND mem ｜reg	メモリ又はレジスタと、定数との論理積をとります。
NOT mem ｜reg	メモリ又はレジスタの、１の補数をとります。
OR reg，mem ｜reg	レジスタと、メモリ又はレジスタの、ビット毎の論理和をとります
OR mem｜reg，reg	メモリ又はレジスタと、レジスタの、ビット毎の論理和をとります
OR mem｜reg，reg	メモリ又はレジスタと定数との、ビット毎の論理和をとります。
RCL mem ｜reg，1	レジスタ又はメモリを、キャリ・フラグを通して１ビット左に回転させます。
RCL mem ｜reg，CL	メモリ又はレジスタを、キャリ・フラグを通して、CLレジスタで与えられるビット数だけ左に回転させます。 
RCR mem ｜reg，1	メモリ又はレジスタを、キャリ・フラグを通して、１ビット右に回転させます。
RCR mem ｜reg，CL	メモリ又はレジスタを、キャリ・フラグを通して、CLレジスタで与えられるビット数だけ右に回転させます。 
ROL mem ｜reg，1	メモリ又はレジスタを、１ビット左に回転させます。
ROL mem ｜reg，CL	メモリ又はレジスタを、CLレジスタで与えられるビット数だけ左に回転させます。 

	
ROR mem ｜reg，1	メモリ又はレジスタを、１ビット右に回転させます。
ROR mem ｜reg，CL	メモリ又はレジスタを、CLレジスタで与えられるビット数だけ右に回転させます。 
SAL mem ｜reg，1	メモリ又はレジスタを、１ビット左へシフトします。下位ビットには、ゼロが入ります。

論理命令とシフト命令

文法	実　行　結　果
SAL mem ｜reg，CL	メモリ又はレジスタを、CLレジスタで与えられるビット数だけ左にシフトします。下位ビットには、ゼロが入ります。 
SAR mem ｜reg，1	メモリ又はレジスタを１ビット右へシフトします。上位ビットにはもとの最上位ビツトの値が詰まります。
SAR mem ｜reg，CL	メモリ又はレジスタをCLレジスタで与えられるビット数だけ右にシフトします。上位ビットにはもとの最上位ビットの値が詰まります 
SHL mem ｜reg，1	メモリ又はレジスタを１ビット左へシフトします。下位ビットにはゼロが入ります。 SHLと SALは別のニ－モニックです。
SHL mem ｜reg，CL	メモリ又はレジスタをCLレジスタで与えられるビット数だけ左へシフトします。下位ビットにはゼロが入ります。 SHLと SALは別のニ－モニックです。
SHR mem ｜reg，1	メモリ又はレジスタを１ビット右へシフトします。上位ビットにはゼロが入ります。

SHR mem ｜reg，CL	メモリ又はレジスタをCLレジスタで与えられるビット数だけ右へシ フトします。上位ビツトにはゼロが入ります。 
TEST reg，mem ｜reg	レジスタと、メモリ又はレジスタとの論理積をとります。結果のい かんによってフラグはセットされるが、受け側のレジスタは変化し ません。
TEST mem｜reg，reg	レジスタ又はメモリと、レジスタとの論理積をとります。結果のい かんによってフラグはセットされるが、受け側のメモリ又はレジス タは変化しません。
TEST mem｜reg，numb	定数とメモリ又はレジスタとの論理積をとります。結果のいかんに よってフラグはセットされるが、メモリ又はレジスタは変化しませ ん。
XOR reg，mem｜reg	リジスタと、メモリ又はレジスタの排他的論理和をとります。
XOR mem ｜reg,reg	メモリ又はレジスタと、レジスタとの排他的論理和をとります。
XOR mem ｜reg，numb	メモリ又はレジスタと、定数との排他的論理和をとります。

D．4　ストリング命令

ストリング命令は１ないし２つのオペランドを持ちます。オペランドはそのオペランドの型を定義し、その型によって操作が１バイトずつ行われたりします。オペランドが２つある場合ソース・オペランド（右）はＳＩレジスタでアドレシングされ、受け側オペランド（右）はＤＩレジスタでアドレシングされます。

アドレシングは必ずＳＩ，ＤＩレジスタで行なわれます。ストリング操作の時には、ＤＩでアドレシングされる受け側オペランドは必ずエクストラ・セグメント（ＥＳ）上になければならないことに注意してください。

ストリング命令

文法	実　行　結　果
CMPS mem｜reg，mem｜reg	受け側からソース・オペランドを引きます。その結果はフラグへ影響するだけです。
LODS mem｜reg	１バイト又は１ワードをソース・オペランドからアキュムレータへ送ります。
MOVS mem｜reg，mem｜reg	１バイト又は１ワードをソースから受け側へ転送します。
SCAS mem｜reg	アキュムレータ（AX又はAL）からオペランドを引きます。その結果はフラグへ影響するだけです。
STOS mem｜reg	アキュムレータからオペランドへ１バイト又は１ワードを送ります

次の表に、ストリング命令のプレフィクスを定義します。プレフィクスがあると、ＣＸレジスタで与える回数だけ、ストリング命令をくり返します。１回くり返されるたびに、ＣＸの値からは１が引かれます。プレフィクスのニーモニックは、ストリング命令のニーモニックの直前に置きます。

プレフィクス命令

文法	結果
REP	CXレジスタがゼロになるまでくり返します。
REPZ	CXレジスタがゼロになるか、ゼロ・フラグ（ZF）がゼロになるまでくり返します。
REPE	REPXと同じ。
REPNZ	CXレジスタがゼロになるか、ゼロ・フラグ（ZF）がゼロでなくなるまでくり返します。
REPNE	REPNZ と同じ。

D．５　制御移動命令

制御移動命令には次の４種類があります。

（１）コール、ジャンプ、リターン
（２）条件付ジャンプ
（３）くり返し制御
（４）割込み

制御移動命令は、プログラムの実行をメモリ上のある新しい位置（別のコード・セグメントへも可）へ渡します。制御の移動には、絶対的に移動するものと、ある決まった条件に従って行われるものとがあります。

次の表に、これらの命令を定義します。条件付ジャンプのところで、「上である」とか「下である」とかは、符号なしの値の間の関係を表わし、「大きい」とか「小さい」とかは、符号付きの値の間の関係を表わします。

制御移動命令

文法	実　行　結　果
CALL label	次の命令のオフセット・アドレスをスタックへプッシュし、ラベルで示されるアドレスへジャンプします。
CALL mem｜reg16	次の命令のオフセット・アドレスをスタックへプッシュし、メモリ又はレジスタの内容によって示されるアドレスへジャンプします。
CALLF label	セグメント・レジスタCSをスタックへプッシュし、次の命令のオフセット・アドレスをそのあとにプッシュして、ラベルで示されるアドレスにジャンプします。
CALLF mem	セグメント・レジスタCSをスタックへプッシュし、次の命令のオフセット・アドレスをそのあとにプッシュして、２ワードのメモリ領域の内容で示されるアドレスへジャンプします。
INT numb 8	フラグ・レジスタをスタックへプッシュし（PUSHF と同じ）、TFとIFをクリアし、256 個の割込みベクトルのひとつを用いて、間接修飾により制御を移します。スタックを３レベル使用します。
INTO	OF（オーバーフロ・フラグ）がセットされているとき、フラグレジスタをスタックへプッシュし（PUSHF と同じ）TFとIFをクリアし、割込みベクタの４番目の要素（10H ）を用いて、関接修飾により制御を移します。OFがセットされていなければ、何も起りません。

文法	実 行 結 果
IRET	以前に割込みによってセーブされたアドレスへリターンします。GSとIPと一緒に、フラグレジスタも以前にストアされていたものを再ロードします。スタックは3レベル戻ります。
JA lab 8	"等しいか下でない" とき又は "上である" ((CF 又は ZF) = 0) のとき、ジャンプします。
JAE lab 8	"下でない" 又は "等しいか上である" (CF = 0) のときジャンプします。
JB lab 8	"下である" 又は "等しいか上でない" (CF = 1) のときジャンプします。
JBE lab 8	"等しいか下である" 又は "上でない" ((CF 又は ZF) = 1) のとき、ジャンプします。
JC lab 8	JBに同じ。
JCXZ lab 8	CXレジスタがゼロならばラベルにジャンプします。
JE lab 8	"等しい" 又は "ゼロである" (ZF = 1) のときジャンプします。
JG lab 8	"等しいか小さくない" 又は "大きい"(((SF xor OF)又は ZF)= 0) のとき、ジャンプします。
JGE lab 8	"小さくない" あるいは "等しいか大きい" ((SF xor OF)= 1) のとき、ジャンプします。
JL lab 8	"小さい" あるいは "等しいか大きくない" ((SF xor OF)= 1) のとき、ジャンプします。
JLE lab 8	"等しいか小さい" 又は "大きくない"(((SF xor OF)又は ZF)= 1) のとき、ジャンプします。
JMP label	ラベルへジャンプします。
JMP mem ｜reg16	メモリ又はレジスタの内容が示すアドレスへジャンプします。
JMPF label	ラベルへジャンプします。ラベルは別のコード・セグメントでもよい。
JMPS lab 8	現在のアドレスから±128 バイト以内にあるラベルへジャンプします。
JNA lab 8	JBE に同じ。
JNAE lab 8	JB　に同じ。
JNB lab 8	JAE に同じ。
JNBE lab 8	JA　に同じ。
JNC lab 8	JNB に同じ。
JNE lab 8	"等しくない" あるいは "ゼロでない" (ZF = 0) のときジャンプします。
JNG lab 8	JLE に同じ。
JNGE lab 8	JL　に同じ。
JNL lab 8	JGE に同じ。
JNLE lab 8	JG　に同じ。
JNO lab 8	"オーバーフローでない" (OF = 0) のときジャンプします。
JNP lab 8	"パリティなし" あるいは "奇数パリティ" のときジャンプします
JNS lab 8	"符号なし" (SF = 0) のときジャンプします。
JNZ lab 8	JNE に同じ。
JO lab 8	"オーバーフロー" (OF = 1) のときジャンプします。
JP lab 8	"パリティ" あるいは "偶数パリティ" (PF=1) のときジャンプします。
JPE lab 8	JP　に同じ。
JPO lab 8	JNP に同じ。
JS lab 8	"符号あり" (SF = 1) のときジャンプします。
JZ lab 8	JE　に同じ。
LOOP lab 8	CXレジスタより1を引き、ゼロにならなければラベルにジャンプします。

LOOPE lab 8	CXレジスタより１を引き、ゼロにならずZFフラグがセットされていればジャンプします。"ゼロでなくなるまでループ" あるいは "等しくなるまでループ" 。
LOOPNE lab 8	CXレジスタより１を引き、ゼロにならずZFフラグがセットされていればジャンプします。"ゼロになるまでループ" あるいは "等しくなるまでループ" 。
LOOPNZ lab 8	LOOPNEに同じ。
LOOPZ lab 8	LOOPE に同じ。

文法	実 行 結 果
RET	以前にCALL命令によってプッシュされたリターン・アドレスへ返えります。スタック・ポインタは２増加します。
RET numb	以前にCALL命令によってプッシュされたリターン・アドレスへ戻ります。スタック・ポインタは２＋numb増加します。
RETF	以前にCALL命令によってプッシュされたリターン・アドレスへ戻ります。スタック・ポインタは４増加します。
RETF numb	以前にCALL命令によってプッシュされたリターン・アドレスへ戻ります。スタック・ポインタは４＋numb増加します。

D．６　制御移動命令

プロセッサ制御命令は、フラグ・レジスタを操作します。さらに、これらの命令のうちのいくつかは、8086CPU と外部装置との同期をとることができます。

プロセッサ制御命令

文法	実 行 結 果
CLC	CFフラグをクリアします。
CLD	DFフラグをクリアします。ストリング命令でのオペランドのポインタは自動増加となります。
CLI	IFフラグをクリアし、外部マスカブル割込みを不可にします。
CMC	CFフラグを反転します。
ESC numb 8, mem ｜reg	実効アドレスを計算し、それをアドレス・バスに配置します。(ESCは8087 "演算用" プロセッサで使われます）その他には何もしません。numb8 は 0～63の値でなくてはなりません。
LOCK	プレフィクス命令です。8087プロセッサは、この後に続く命令の実行中 "バス・ロック" 信号を出し続けます。 LOCKプレフィクスはどの命令の前に置いてもよく、バス・ロックはプロセッサがバスを使用するのを妨げる効果があります。これは、資源を分割して使用するのに有効です。
HLT	8086プロセッサを、割込みがかかるまで停止状態にします。
STC	CFフラグをセットします。
STD	DFフラグをセットします。ストリング命令でのオペランドのポインタは自動減少となります。
STI	IFフラグをセットし、外部マスカブル割込みを可能にします。
WAIT	"テスト" 端子に信号が来ていなければ、8086プロセッサを "ウエイト" 状態にします。

付録Ｅ：各レジスタの特長

ＡＸ：アキュムレ－タ・レジスタ・・・	転送および演算に用いられる。ＡＸレジスタを用いることにより命令のバイト数が短かくなるものや、実行時間が短かくなるものがある。また、必ずＡＸ（ＡＬ）レジスタを使用しなければならない命令もある（乗除算における、被乗除算の格納、符号拡張命令などに使用）。
ＢＸ：ベ－ス・レジスタ・・・・・・・	転送および演算に用いられる。アドレスの間接指定に使用することができる。
ＣＸ：カウンタ・レジスタ・・・・・・	転送および演算に用いられる。ル－プ命令、ストリング命令におけるくり返し回数の自動スカウントに用いられる。ＣＬレジスタとして、論理演算命令のシフト数指定に用いられる。
ＤＸ：データ・レジスタ・・・・・・・	転送および演算に用いられる。一部の演算命令において、ＡＸレジスタと対で使用される（乗除算および符号拡張命令）。
ＳＰ：スタック・ポインタ・・・・・・	スタックの最新のデ－タの格納されているアドレスを示す。スタック操作（ＰＵＳＨ，ＰＯＰ命令・ＣＡＬＬ，ＲＥＴ命令など）を行なうと、自動的に値が増減される。転送および演算にも使用可能であるが、新規にスタック。ポインタを設定する場合以外には行うべきではない。
ＢＰ：ベ－ス・ポインタ・・・・・・・	スタック上のデ－タのアドレスの間接指定に用いる。ユ－ザが自由に変更できるので、スタックを取り出すのに使用される。他の領域のデ－タの間接指定や転送および演算にも用いられる。
ＳＩ：ソ－ス・インデックス・・・・・	アドレスの間接指定に用いる。特にストリング命令においては、メモリからレジスタにデ－タ転送（ロ－ド）するための転送先（ソ－ス）アドレスの指定に用いられる。一般の転送や演算にも用いられる。
ＤＩ：ディスティネ－ション・・・・・ インデックス	アドレスの間接指定に用いる。特にストリング命令においては、レジスタからメモリにデ－タを転送（ストア）するための転送先（ディスティネ－ション）アドレスの指定に用いられる。一般の転送や演算にも用いられる。

ＣＳ：コ－ド・セグメント・・・・・・	ＣＰＵが実行すべき命令が格納されているセグメントのはじめを示す。
ＤＳ：データ・セグメント・・・・・・	データの格納されているセグメントのはじめを示す。
ＳＳ：スタック・セグメント・・・・・	スタック動作（後述）の行われるセグメントのはじめを示す。
ＥＳ：エクストラ・セグメント・・・・	データ転送命令において、デ－タの転送先などに用いられるセグメントのはじめを示す。

※これらのレジスタのうち、特に汎用レジスタの機能について、次の表にまとめておきます。

	加減乗除	アドレス指定	ストリング命令	ループカウント	シフト回転	1／0命令
AX	o					o*3
BX	o	o				
CX	o			o	(CL)	
DX	o*1					o*3
SP	o	△*2				
BP	o	o				
SI	o	o	o			
DI	o	o	o			

◆汎用レジスタの機能

＊１　乗除算では、被乗除数はＡＬ，ＡＨまたはＡＸ，ＤＸレジスタペアに入れられる。

＊２　ＳＰのＰＵＳＨ，ＰＯＰ命令において、最新データの置かれているアドレスを指定するのに自動的に使用されるが、ユーザーは直接にＳＰを用いたアドレス指定を行うことはできない。

＊３　Ｉ／Ｏ命令では、ＡＸ（ＡＬ）レジスタがデータの受け渡しに、ＤＸレジスタが、Ｉ／Ｏポート番号の指定に用いられる。

付録Ｆ：リンク　情報

ＳＨＡＬではモジュールやセグメントが何個あってもかまいませんが、いくつかの情報によってリンクする時どのように組み合せてメモリ上に配置するか、或いはセグメントの順番などを決めます。

Ｆ．１　アラインメント

セグメントやモジュールのベースアドレスをメモリ中のどのような位置に置くかを指定します。

これは　ＢＹＴＥ
　　ＷＯＲＤ
　　ＰＡＲＡ
　　ＰＡＧＥ　の４種類があり、それぞれアドレスが１、２、１６、２５６、バイトで割り切れる位置という意味を持ちます。

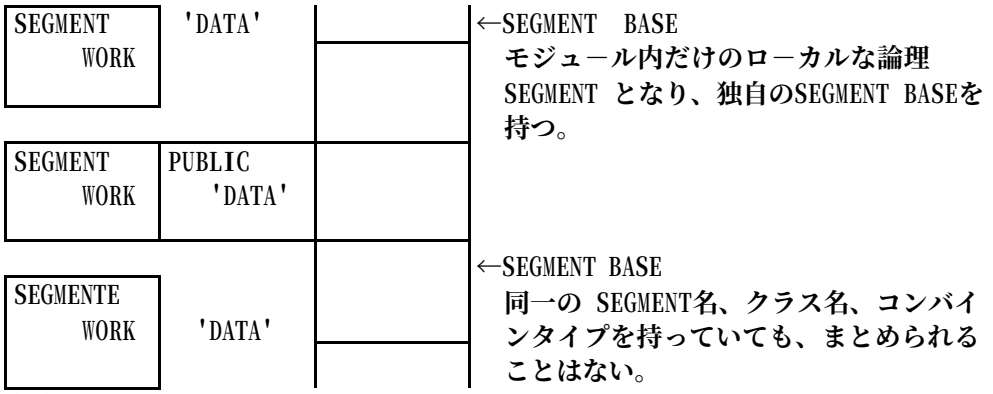
ＢＹＴＥ　アドレスは　ｘｘｘｘｘＨ　（どこでも置けます）
ＷＯＲＤ　アドレスは　ｘｘｘｘｅＨ　（ｅは偶数、最下位ビットは０）
ＰＡＲＡ　アドレスは　ｘｘｘｘ０Ｈ　（下位４ビットは０）
ＰＡＧＥ　アドレスは　ｘｘｘ００Ｈ　（下位バイトは０）

Ｆ．２　組み合わせ

どのように組み合せてメモリに配置するかを設定します。

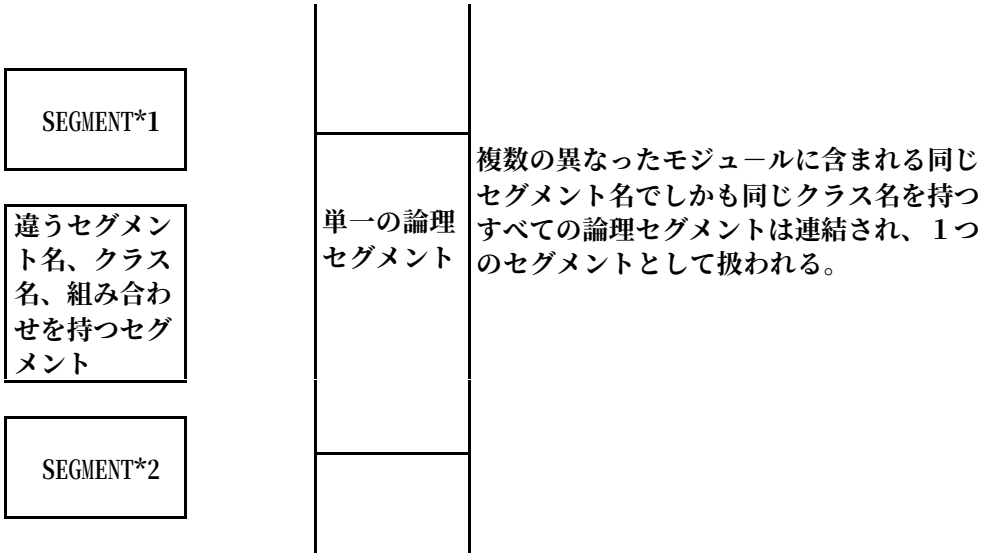
（１）　ＰＲＩＶＡＴＥ　（無指定）

他のセグメントとは別のものとして扱われ、そのモジュール内だけのローカルな論理セグメントとなります。



（２）　ＰＵＢＬＩＣ

他のモジュールに含まれる同じ名前とクラス名を持つ、すべての論理セグメントと連結され、１つの論理セグメントとして扱われます。



ほかに

- ＣＯＭＭＯＮ
- ＳＴＡＣＫ
- ＭＥＭＯＲＹ　がありますが説明は省略します。

付録Ｇ：レコード宣言・構造体宣言

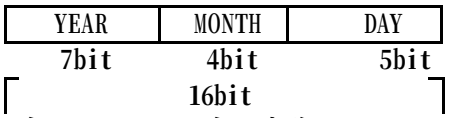
Ｇ．１　レコード宣言

```
RECORD        レコード名
  field名     BIT    ビット幅 [VALUE 定数式 ]
    ・
    ・
END          (レコード名)
```

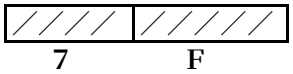
レコード宣言は、１６ビットをｂｉｔ単位に分割して、複数個のフィールドを定義し、それぞれを変数として使用できるようにするものです。

例） ＲＥＣＯＲＤ　　ＤＡＴＥ
　　ＹＥＡＲ　　ＢＩＴ　　７
　　ＭＯＮＴＨ　　ＢＩＴ　　４
　　ＤＡＹ　　ＢＩＴ　　５
　　ＥＮＤ　　（ＤＡＴＥ）

このとき各フィールドは次のようになります。



レコード名とフィールド名は省略することができません。
ビット幅は合計が１６ビットより大きい場合は、エラーとなります。
ＶＡＬＵＥの定数は、初期値またはデフォールトの値で、フィールドが持っている
ビット幅に入る値を指定する必要があります。
上記例のＹＥＡＲは長さが７ビットですから、１６進の７Ｆまで定義できます。



領域の確保は、次のようにします。

レコード名　　変数名 ［＝〈初期値の並び〉］

この場合、変数は次の属性を持ちます。
・ＳＥＧＭＥＮＴ　：割り当てられた時点の論理ＳＥＧＭＥＮＴ名
・ＯＦＦＳＥＴ　　：変数の先頭バイト位置のオフセット
・型　　：そのレコードの型

Ｇ．２　構造体宣言

STRUCT　構造体名
　型　フィールド名［(n)］
　：　　・　［＝初期値］
　　・
　　・
END　（構造体名）

構造体宣言は構造体の構成を宣言し、構造体名や構造体中のフィールド名を使用できるようにするものです。

例）　ＳＴＲＵＣＴ　　ＢＯＸ
　　ＷＯＲＤ　　Ｘ＿ＶＡＬＵＥ
　　ＷＯＲＤ　　Ｙ＿ＶＡＬＵＥ
　　ＷＯＲＤ　　Ｚ＿ＶＡＬＵＥ
　　ＥＮＤ　　（ＢＯＸ）

← WORD →
Ｘ＿ＶＡＬＵＥ
Ｙ＿ＶＡＬＵＥ
Ｚ＿ＶＡＬＵＥ

構造体ＢＯＸの構成

構造体としてＢＯＸを持つ領域、ＢＯＸ１の定義は下記の通りに行います。
　ＢＯＸ　　ＢＯＸ１　＝　〈１０，２０，３０〉
それは、ＢＯＸという型紙の領域を確保することで、メモリへの割り当て、および初期化が行なわれます。

← WORD →
１０
２０
３０

構造体ＢＯＸ１

このように定義されたＢＯＸ１のＸ＿ＶＡＬＵＥ，Ｙ＿ＶＡＬＵＥの参照は次のように行ないます。
データ構造中の各フィールド構造体のベースアドレスを示す〈ａｄｄｒｅｓｓ式〉および宣言された〈フィールド名〉を使用して、次のように表現することができます。
　　〈アドレス式〉.〈フィールド名〉

例）　ｍｏｖ　ｄｘ，ｂｏｘ１．ｘ＿ｖａｌｕｅ

このような表現を用いた場合、それぞれの属性は次のようになります。

・ＯＦＦＳＥＴ　：　〈アドレス式〉の持つＯＦＦＳＥＴ属性＋構造体宣言で定義されフィールドのオフセット

・ＳＥＧＭＥＮＴ：　〈アドレス式〉の持つＳＥＧＭＥＮＴ属性

・型　　：　〈フィールド名〉の持つ型

付録Ｈ：ＰＵＳＨとＰＯＰについて

・ＳＡＶＥのために便利なＰＵＳＨ，ＰＯＰを紹介します。

ＰＵＳＨ	レジスタ セグメント　レジスタ ＷＯＲＤのメモリー

１６ビットのレジスタあるいは、セグメント　レジスタ、あるいは、メモリーの値をスタックのトップにＰＵＳＨする

ＰＯＰ	レジスタ セグメント　レジスタ ＷＯＲＤのメモリー

１６ビットのレジスタなどを、スタックのトップからＰＯＰする。

※ＰＵＳＨ／ＰＯＰは、ｌａｓｔ　ｉｎ，　ｆｉｒｓｔ　ｏｕｔです。

一番最後にＰＵＳＨしたものから、ＰＯＰすると元通りになります。

```
        PUSH        AX
|       PUSH        SI
||      PUSH        CX
|||     DISPLAY     ("SANPLE %d %d¥n", CX , SI)
||      POP         CX
|       POP         SI
        POP         AX
```

このほかにも、レジスタの値を一時的にＳＡＶＥする時などにも使いますが、

このＰＵＳＨ，ＰＯＰを使用する時には注意が必要です。